INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fences in Weak Memory Models

Jade Alglave — Luc Maranget

N° 7010

Juillet 2009

ISSN 0249-6399 ISRN INRIA/RR--7010--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Fences in Weak Memory Models

Jade Alglave , Luc Maranget

Thème :
Équipe-Projet Moscova

Rapport de recherche n° 7010 — Juillet 2009 — 39 pages

Abstract: We present here an axiomatic framework, implemented in the Coq proof assistant, for defining weak memory models in terms of several parameters: local reorderings of reads and writes, and visibility of inter and intra processor communications through memory. In this context, we provide formal definition of weak memory models induced by architectures, illustrated by definitions of SC and Sparc TSO. Moreover, we define a comparison over architectures, an architecture A_1 being weaker than another one A_2 when A_1 allows more behaviours than A_2. In addition, we provide a characterisation of behaviours allowed by A_1 which are also valid on A_2. By that means, we provide a simple characterisation of SC and TSO behaviours on any weaker architecture. We also provide an abstract notion of what should be the action and placement of fences to restore a given model from a weaker one.

Key-words: Weak Memory Models, Fences

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Une étude de l'action des barrières au sein de modèles de mémoire relâchés

Résumé : Nous proposons un environnement générique, implémenté au sein de l'assistant de preuve Coq, pour définir des modèles de mémoire relâchés en fonction de plusieurs paramètres: réordonnancements de lectures et écritures, et visibilités des communications via la mémoire. Dans ce contexte, nous fournissons une définition formelle d'un modèle de mémoire induit par une architecture, que nous illustrons par les définitions de SC et Sparc TSO. Par ailleurs, nous définissons une notion de comparaison de deux architectures, une architecture A_1 étant considérée plus faible qu'une architecture A_2 si A_1 autorise plus de comportements que A_2. De plus, nous fournissons une caractérisation des comportements autorisés par A_1 qui sont également valides au sein de A_2, ce qui nous permet de donner une caractérisation simple de SC et TSO sur des architectures plus faibles. Nous fournissons également une formalisation du pouvoir et du placement des barrières mémoires pour restaurer un modèle donné depuis un modèle plus faible.

Mots-clés : Modèles de mémoires relâchés, Barrières

Contents

1 Introduction

Memory models are what describe and constrain the behaviour of a program running on a multiprocessor. That said, understanding what a program would do on such a machine requires a precise definition of the memory model induced by the machine, that is, the underlying memory system and the behaviour of the processors involved. Previous studies [14, 18] have discussed the need for a rigorous definition of weak memory models, which some of the public documentations [3, 4] lack. We provide here a generic and axiomatic framework to precisely define a memory model in terms of several parameters and test it against real hardware.

Let us consider a *shared-memory multiprocessor* system, that consists of several *processors* writing to or reading from a common *shared memory*. We will discuss here what representation of memory and processor behaviour we consider.

Representation of memory One representation of a shared memory could be a single memory on which several processors operate simultaneously, all their writes being commited to memory as soon as they are issued. Thus, one can consider the connection between processors and memory as direct: as soon as one processor writes to memory, the value written overwrites the previous value and is immediately available to all processors. This property, called *store atomicity*, has been examined and advocated as valuable [16, 8, 11] as it provides the guarantee that actions on such a memory are serialisable, which leads to a rather understandable memory model. However, it is not guaranteed on several real architectures [1, 10, 3, 4]. They indeed relax the store atomicity constraint, which means a write is not available to all processors at once. For example a write is at first initiated by a given processor, then committed to a cache, and finally to memory. This last step is sometimes called *globally performed* [15]. Even without assuming writes to be committed immediately, we suppose a total order on the globally performed writes to the same location, a property sometimes called *coherence* [5] that is widely assumed by modern architectures [1, 10, 3, 4].

Processor behaviour One representation of a processor behaviour could suppose a sequential order, consistent with the program order, of all the reads and writes events issued by a given processor, as a generalisation of the uniprocessor case. However, modern architectures [1, 10, 3, 4] provide *relaxed memory models* that do not constrain the way reads and writes are ordered that much. These constraints, or their relaxation, are often gathered behind the term *instruction reordering* [8, 11].

1.1 An axiomatic generic model

We will precisely define an *architecture* in terms of its ordering and store atomicity relaxations at section 2.1.3. For example, Sequential Consistency (henceforth *SC*) [17], supposes writes to be committed to memory as soon as they are issued, and that the program order is maintained between all accesses, thus being the strongest (in a sense that will be defined precisely at section 2.2) memory model. We will illustrate how to instanciate our model to produce *SC* and Sparc *TSO*

P_0	P_1
(a) `x` $\leftarrow 1$	(c) `y` $\leftarrow 1$
(b) `r2` $\leftarrow$ `y`	(d) `r4` $\leftarrow$ `x`
i3 $r2 = 0 \wedge r4 = 0$?	

Figure 1: i_3 exhibits non-SC behaviour on modern architectures

[1], and show equivalence with the native models, together with characterisation of executions that would be valid on these models.

1.2 Study of barriers power

SC provides indeed a rather comfortable programming model, which explains why most architectures provide mechanisms such as barriers and locks, to restore it from a weaker model. However, it is not clear how much power a barrier needs to provide the illusion of *SC*, and where to place these constructions in the code. We examine this question at section 3.1 from a general point of view: we provide a sufficient condition on barriers to restore a strongest model from a weaker one. Moreover, we refine this condition in some particular yet interesting cases at section 3.2, such as *TSO* [1].

1.3 Case study: a Power model

Our generic framework, implemented in the Coq proof assistant [12], has two companion tools: memevents, written in OCaml, which is an exact implementation of our axiomatic model, and litmus, which runs the same inputs that memevents takes on real hardware.

We provide a serie of tests to instanciate properly our model with respect to a given machine or architecture, which allowed us to design a model for a significant fragment of the Power architecture with barriers.

2 Description of the model

2.1 Axiomatisation

The classical test depicted at fig. 1, which can be found in [6] with number 2.3a illustrates the fact that we cannot use an interleaving semantics to reason on executions induced by weak memory models, as its exhibits a non-SC behaviour on some current architectures [6, 5]. Instead, we reason on relations over read and write events raised by an instruction.

2.1.1 Basic objects

An event is an abstraction of a memory access performed during the execution of a multiprocessor program. We note $\mathfrak{E}$ the set of events generated by a particular execution. Events are of two kinds: reads and writes, which sets will be depicted by $\mathfrak{R}$ and $\mathfrak{W}$. Henceforth, we will note e for an event, r for a read, and w for a write. An event e will hold its direction (R or W), its location, given by $loc\ e$,

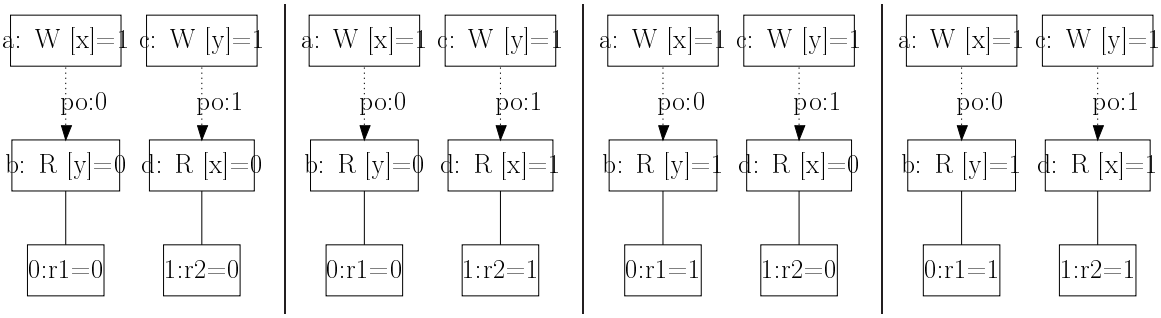


Figure 2: Event structures for test **i3**.

its value, given by *val* e and its processor, given by *proc* e. We will note (a) `x` $\leftarrow$ `v` for a write to location x with value v labelled (a), and (b) `r1` $\leftarrow$ `y`, for a read from y labelled (b).

An execution is also characterised by the *program order* $\overset{\text{po}}{\rightarrow}$, a relation on events that reflects the sequential execution of instructions on a single processor: given two instructions i_1 and i_2 that generate events e_1 and e_2, having $e_1 \overset{\text{po}}{\rightarrow} e_2$ on events simply means that i_1 precedes i_2 in program order. $\overset{\text{po}}{\rightarrow}$ is a total order amongst the events from the same processor[1] and never relates events from different processors.

We collect these informations into an *event structure*, depicted by E:

$$E \triangleq (\mathfrak{E}, \overset{\text{po}}{\rightarrow})$$

Figure 2 illustrates the event structures associated to the test **i3** depicted at fig. 1.

2.1.2 Execution witnesses

We postulate two relations over events: $\overset{\text{rf}}{\rightarrow}$ and $\overset{\text{ws}}{\rightarrow}$.

Rf A *read-from map*, links a read event with the write event that provides its value. We represent the notion by a relation from writes to reads, which is well-formed in the following sense:

$$\begin{array}{rcl} \overset{\text{prf}}{\rightarrow} & \triangleq & \{(w,r) \mid \exists lv, w \in \mathfrak{W}_{l,v} \wedge r \in \mathfrak{R}_{l,v}\} \\ wf_{rf}\ \overset{\text{rf}}{\rightarrow} & \triangleq & \overset{\text{rf}}{\rightarrow} \subseteq \overset{\text{prf}}{\rightarrow} \wedge \forall r, \exists !\, w, w \overset{\text{rf}}{\rightarrow} r \end{array}$$

We gathered first all pairs of writes and reads with same location l and value v, which sets are depicted by $\mathfrak{W}_{l,v}$ and $\mathfrak{R}_{l,v}$, and then enforced the uniqueness of read sources.

[1] When some instructions may perform several memory accesses, $\overset{\text{po}}{\rightarrow}$ should include some of intra-instruction dependencies [18], thus becoming a partial order on events from a same processor.

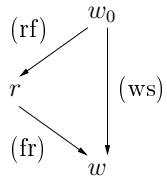
Ws The *write serialisation* is a total order of the writes to a same location. Thus, we first gather all pairs of writes to the same location, and we require the relation to be a total order on writes to a same location l, which set will be depicted by $\mathfrak{W}_l$:

$$\begin{array}{rcl} \overset{\text{pws}}{\rightarrow} & \triangleq & \{(w_1, w_2) \mid \exists l, w_1 \in \mathfrak{W}_l \wedge w_2 \in \mathfrak{W}_l\} \\ wf_{ws}\ \overset{\text{ws}}{\rightarrow} & \triangleq & \overset{\text{ws}}{\rightarrow} \subseteq \overset{\text{pws}}{\rightarrow} \wedge \forall \ell, \mathit{total\ order}\ (\overset{\text{ws}}{\rightarrow} \restriction \mathfrak{W}_\ell)\ \mathfrak{W}_\ell \end{array}$$

The notation $\overset{\text{ws}}{\rightarrow} \restriction \mathfrak{W}_\ell$ stands for the restriction of the relation $\overset{\text{ws}}{\rightarrow}$ to the set $\mathfrak{W}_\ell$, *i.e.* $\overset{\text{ws}}{\rightarrow} \cap (\mathfrak{W}_\ell \times \mathfrak{W}_\ell)$.

Fr From these two relations, we deduce a third one, *fr*:

$$r \overset{\text{fr}}{\rightarrow} w \triangleq \exists\ w'\ , w' \overset{\text{rf}}{\rightarrow} r \wedge w' \overset{\text{ws}}{\rightarrow} w$$



As we said, $\overset{\text{ws}}{\rightarrow}$ orders globally performed writes to the same location; thus, if a write w' is before another write w in $\overset{\text{ws}}{\rightarrow}$, we know that w' is globally – that is, for every processor – before w. Furthermore, if a read r reads from w', we consider r to be globally ordered the following write w: otherwise, there would be no guarantee r actually read its value from w', thus contradicting the $\overset{\text{rf}}{\rightarrow}$ relation between them.

Execution witnesses We gather these relations — except $\overset{\text{fr}}{\rightarrow}$ as it can be deduced from the others — into an *execution witness*, depicted by X:

$$X \triangleq (\overset{\text{rf}}{\rightarrow}, \overset{\text{ws}}{\rightarrow})$$

Figure 3 adds $\overset{\text{rf}}{\rightarrow}$ and $\overset{\text{fr}}{\rightarrow}$ edges to the event structures of figure 2. There are no $\overset{\text{ws}}{\rightarrow}$ edges among the (non-initialisation) writes shown. However, we see some $\overset{\text{fr}}{\rightarrow}$ arrows which follow from the serialization of init stores (which come first in $\overset{\text{ws}}{\rightarrow}$) and of stores generated by instructions. For instance, in the leftmost picture, we have $d \overset{\text{fr}}{\rightarrow} a$. Indeed, the load d reads the initial value of location x, which location is overwritten (later!) by the store a.

We have the associated well formedness predicate wf, being the conjunction of the predicates for $\overset{\text{rf}}{\rightarrow}$ and $\overset{\text{ws}}{\rightarrow}$.

Initial and final states The write serialization provides a natural way to define the initial and final states of an execution:

$$\begin{array}{rcl} \mathit{init}\ X & \triangleq & \{w \mid \neg(\exists w', w' \overset{\text{ws}}{\rightarrow} w)\} \\ \mathit{final}\ X & \triangleq & \{w \mid \neg(\exists w', w \overset{\text{ws}}{\rightarrow} w')\} \end{array}$$

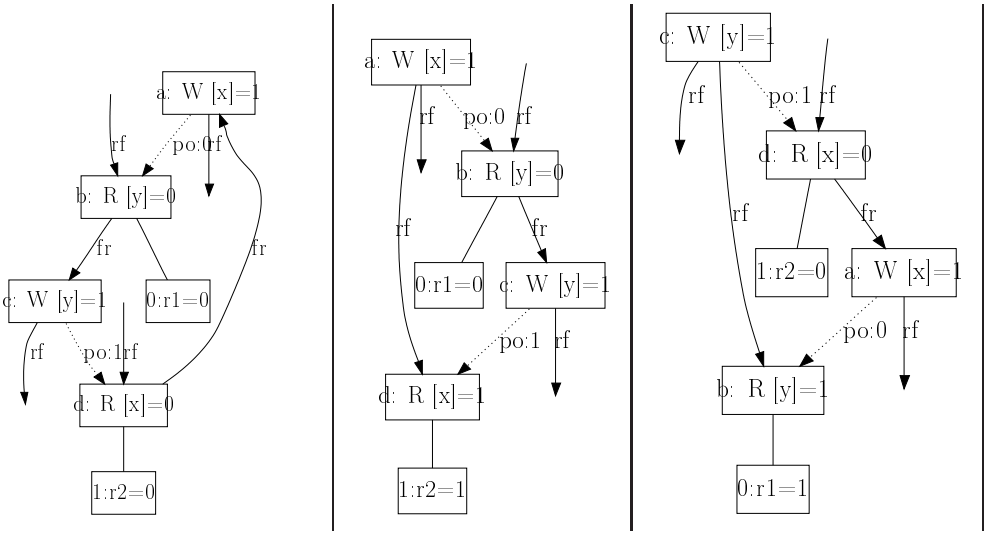


Figure 3: Execution witnesses for **i3**.

2.1.3 Architectures

We define here what we consider to be an *architecture*.

Preserved program order We assume a function *ppo*, which gathers all pairs of events that are not to be reordered with respect to the program order $\xrightarrow{\text{po}}$. Consider for example the test **i3**, depicted at fig. 1: the specified outcome would be valid only if writes and reads to different locations could be reordered. Thus, an architecture that would authorise the specified outcome would not include write-read pairs in its preserved program order.

We will note $\xrightarrow{\text{ppo}}$ for the relation outputed by this function on a given event structure E, which is to be included in $\xrightarrow{\text{po}}$. This relation is to be considered global, that is, all processors must behave with respect to the constraints induced by it.

Globality of relations As stated in the introduction, we consider writes to be non-atomic, that is, not necessarily available to all parts of the memory system at once. Thus, the behaviour of all processors must not necessarily include the constraints induced by $\xrightarrow{\text{rf}}$ relations. However, we distinguish the constraints induced by internal $\xrightarrow{\text{rf}} - \xrightarrow{\text{rf}}$ relation on a same processor – and external $- \xrightarrow{\text{rf}}$ from one processor to another. Thus, we split the $\xrightarrow{\text{rf}}$ relation into $\xrightarrow{\text{rfi}}$, which represents the events in $\xrightarrow{\text{rf}}$ on the same processor, and $\xrightarrow{\text{rfe}}$, which represents the events in $\xrightarrow{\text{rf}}$ on different processors:

$$w \xrightarrow{\text{rfi}} r \triangleq w \xrightarrow{\text{rf}} r \wedge \mathit{proc}\ w = \mathit{proc}\ r$$

$$w \xrightarrow{\text{rfe}} r \triangleq w \xrightarrow{\text{rf}} r \wedge \mathit{proc}\ w \neq \mathit{proc}\ r$$

Relations induced by the presence of barriers We assume given a function ab, which, provided an event structure E and an execution witness X, defines the relation over events induced by the presence of a barrier in between – in $\overset{\text{po}}{\rightarrow}$ – two instructions:

$$ab : \mathcal{E} \rightarrow \mathcal{X} \rightarrow rln\ \mathfrak{E}$$

where $\mathcal{E}$ (resp. $\mathcal{X}$) is the type of event structures (resp. execution witnesses). These informations are what defines for us an architecture, depicted by A:

Definition 1 (Architecture)

$$A \triangleq (ppo, int, ext, ab)$$

2.1.4 Validity of an execution with respect to an architecture

We define here what it means for an execution witness X to be valid on a given architecture A.

Uniprocessor behaviour Some documentations [3] claim that a sole processor is supposed to respect the *sequential execution model*, that is:

> the model of program execution in which the processor appears to execute one instruction at a time, completing each instruction before beginning to execute the next instruction

Following Alpha [10], we define the *processor issue order*, depicted by the $\overset{\text{pio}}{\rightarrow}$ relation, as follows:

$$e_1 \overset{\text{pio}}{\rightarrow} e_2 \triangleq e_1 \overset{\text{po}}{\rightarrow} e_2 \wedge loc\ e_1 = loc\ e_2$$

We call $\overset{\text{hb}}{\rightarrow}$ the union of the three relations $\overset{\text{rf}}{\rightarrow}$, $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{fr}}{\rightarrow}$:

$$\overset{\text{hb}}{\rightarrow} \triangleq \overset{\text{rf}}{\rightarrow} \cup \overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow}$$

Notice that $\overset{\text{hb}}{\rightarrow}$ is not the proper happens-before relation in the general case, but rather the happens-before of a memory with multi-copy-atomic writes. We define the general happens-before relation in the next section.

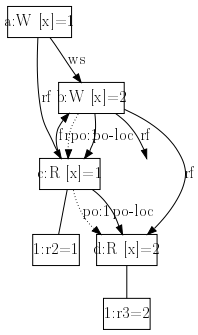
To provide our executions the guarantee that they respect the sequential execution model, we require that all the relations $\overset{\text{rf}}{\rightarrow}$, $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{fr}}{\rightarrow}$ are consistent with the processor issue order, that is:

$$uniproc \triangleq acyclic\ (\overset{\text{hb}}{\rightarrow} \cup \overset{\text{pio}}{\rightarrow})$$

Figure 4 gives an example of an outcome that is forbidden because of *uniproc*.

There are two executions for this outcome, with different write serializations: $a \overset{\text{ws}}{\rightarrow} b$ on the left, and $b \overset{\text{ws}}{\rightarrow} a$ on the right. In former case, we have $c \overset{\text{fr}}{\rightarrow} b$ (by $a \overset{\text{ws}}{\rightarrow} b$ and $a \overset{\text{rf}}{\rightarrow} c$). Thus, invalidation follows from cycle $b \overset{\text{pio}}{\rightarrow} c \overset{\text{fr}}{\rightarrow} b$. In the latter case, the cycle is $a \overset{\text{rf}}{\rightarrow} c \overset{\text{pio}}{\rightarrow} d \overset{\text{fr}}{\rightarrow} a$, the last step following from $b \overset{\text{ws}}{\rightarrow} a$ and $b \overset{\text{rf}}{\rightarrow} d$.

P_0	P_1
(a) `x` $\leftarrow$ `1`	(b) `x` $\leftarrow$ `2`
	(c) `r2` $\leftarrow$ `x`
	(d) `r3` $\leftarrow$ `x`
Forbidden: `1:r2=1; 1:r3=2;`	



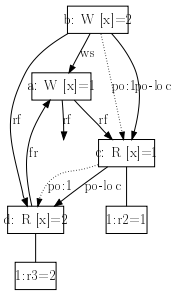


Figure 4: Invalid executions by *uniproc*.

All together We call $\overset{\text{ghb}}{\rightarrow}$ the union of the relations that are global:

$$\overset{\text{ghb}}{\rightarrow} \triangleq \overset{\text{ppo}}{\rightarrow} \cup \overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{rf}\,?}{\rightarrow} \cup \overset{\text{ab}}{\rightarrow}$$

with $\overset{\text{rf}\,?}{\rightarrow} \triangleq \overset{\text{rfi}\,?}{\rightarrow} \cup \overset{\text{rfe}\,?}{\rightarrow}$ where $\overset{\text{rfi}\,?}{\rightarrow}$ (resp. $\overset{\text{rfe}\,?}{\rightarrow}$) is $\overset{\text{rfi}}{\rightarrow}$ (resp. $\overset{\text{rfe}}{\rightarrow}$) if *int* (resp. *ext*) is *true*, the empty relation otherwise.

We can now define what a valid execution is, with respect to an architecture A:

Definition 2 (Valid execution)

$$A.valid\ E\ X \quad \triangleq \quad wf \wedge uniproc \wedge acyclic \ \ (\overset{ghb}{\rightarrow})$$

Weak Memory Models Let $\mathcal{W}$ be the type of memory models, defined as follows:

$$\mathcal{W} \triangleq \mathcal{E} \rightarrow \mathcal{X} \rightarrow \{\top, \bot\}$$

Thus, we defined a function $Wmm - \mathcal{A}$ being the type of architectures, which produces a weak memory model induced by A:

Definition 3 (Weak Memory Model)

$$\begin{array}{rcl} Wmm & : & \mathcal{A} \rightarrow \mathcal{W} \\ Wmm(A) & \triangleq & \forall\ E\ X, A.valid\ E\ X \end{array}$$

Henceforth, we will note $AWmm$ for $Wmm(A)$.

2.1.5 Examples

We will show how to produce a particular model from our generic framework on two classical memory models, *Sequential Consistency* [17], later on referred to as *SC* and *TSO* [1], thus illustrating the concepts we used to define our framework. We will show at section 2.3 that these definitions are equivalent to the native ones.

Sequential consistency *SC* has been defined by Lamport as follows:

> The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. [17]

We give here a formal definition of an *SC* execution. We need at first a *sequential execution* $\overset{\text{ex}}{\rightarrow}$, that is a total order consistent with the program order:

$$seq \overset{\text{ex}}{\rightarrow} \triangleq total\ order \overset{\text{ex}}{\rightarrow} \mathfrak{E} \wedge \overset{\text{po}}{\rightarrow} \subseteq \overset{\text{ex}}{\rightarrow}$$

We need to highlight the implicit execution model, which states that a read r reads from the most recent write that is before it – in $\overset{\text{ex}}{\rightarrow}$. Let us note $pw_o(r)$ the set of previous writes for r in a partial order o to define the $\overset{\text{rf}}{\rightarrow}$ relation for an *SC* execution – that is, which read reads from which write:

$$SC.rf \overset{\text{ex}}{\rightarrow} \triangleq \{(w,r) \mid w \overset{\text{prf}}{\rightarrow} r \wedge w = \max pw_{\overset{\text{ex}}{\rightarrow}}(r)\}$$

Thus a valid *SC* execution will be given by a sequential execution $\overset{\text{ex}}{\rightarrow}$ and the calculation of its induced $\overset{\text{rf}}{\rightarrow}$ relation as above.

From such an execution, we can produce an execution witness:

$$\begin{aligned} SC.ws \overset{\text{ex}}{\rightarrow} &\triangleq \{(w_1,w_2) \mid w_1 \overset{\text{ex}}{\rightarrow} w_2 \wedge w_1 \overset{\text{pws}}{\rightarrow} w_2)\} \\ SC.wit \overset{\text{ex}}{\rightarrow} &\triangleq (SC.rf \overset{\text{ex}}{\rightarrow}, SC.ws \overset{\text{ex}}{\rightarrow}) \end{aligned}$$

We propose here an alternative notion of *SC*, which we will show equivalent to the native one in cor. 3:

$$\begin{aligned} Sc.Arch &\triangleq (\overset{\text{po}}{\rightarrow}, true, true, \overset{\text{ab}}{\rightarrow}) \\ Sc.Wmm &\triangleq Wmm(Sc.Arch) \end{aligned}$$

TSO To design a proper TSO execution, we need to require what the Sparc documentation [1] specifies:

$$\begin{aligned} R* &\triangleq \{(r,e) \mid r \overset{\text{po}}{\rightarrow} e\} \\ WW &\triangleq \{(w_1,w_2) \mid w_1 \overset{\text{po}}{\rightarrow} w_2\} \\ ptso \overset{\text{ex}}{\rightarrow} &\triangleq partial\ order \overset{\text{ex}}{\rightarrow} \mathfrak{E} \wedge \\ &\quad R* \subseteq \overset{\text{ex}}{\rightarrow} \wedge \\ &\quad WW \subseteq \overset{\text{ex}}{\rightarrow} \wedge \\ &\quad \exists \overset{\text{tso}}{\rightarrow}, \overset{\text{tso}}{\rightarrow} \subseteq \overset{\text{ex}}{\rightarrow} \wedge \\ &\quad total\ order \overset{\text{tso}}{\rightarrow} \mathfrak{W} \end{aligned}$$

Moreover, we need to highlight the explicit execution model, provided by the Val axiom in the documentation:

$$Val(L_a) = Val(\max_{\overset{\text{ex}}{\rightarrow}}\{S_a \mid S_a \overset{\text{ex}}{\rightarrow} L_a \vee S_a \overset{\text{po}}{\rightarrow} L_a\})$$

which states that a read r reads from the most recent write that is before it in $\overset{\text{ex}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$. Thus we define the $\overset{\text{rf}}{\rightarrow}$ relation for a TSO execution – that is, which read reads from which write:

$$TSO.rf \overset{\text{ex}}{\rightarrow} \triangleq \{(w,r) \mid w \overset{\text{prf}}{\rightarrow} r \wedge w = \max(pw_{(\overset{\text{ex}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow})}(r))\}$$

As in the SC case, we produce an execution witness:

$$\begin{array}{rcl} TSO.ws \overset{\text{ex}}{\rightarrow} & \triangleq & \{(w_1,w_2) \mid w_1 \overset{\text{pws}}{\rightarrow} w_2 \wedge w_1 \overset{\text{ex}}{\rightarrow} w_2\} \\ TSO.wit \overset{\text{ex}}{\rightarrow} & \triangleq & (TSO.rf \overset{\text{ex}}{\rightarrow} , TSO.ws \overset{\text{ex}}{\rightarrow}) \end{array}$$

We propose here an alternative notion of TSO, which we will show equivalent to the native one in cor. 4:

$$\begin{array}{rcl} ppo\_tso & \triangleq & R* \cup WW \\ Tso.Arch & \triangleq & (ppo\_tso, false, true, ab) \\ Tso.Wmm & \triangleq & Wmm(Tso.Arch) \end{array}$$

The $\overset{\text{ppo}}{\rightarrow}$ is quite clear from the documentation. The Val axiom indicates that the internal $\overset{\text{rf}}{\rightarrow}$ are not included in $\overset{\text{ex}}{\rightarrow}$, whereas the external are, as the write from which a read reads is the max of its previous writes in $\overset{\text{ex}}{\rightarrow}$. Thus we consider $\overset{\text{rfe}}{\rightarrow}$ to be global, whereas $\overset{\text{rfi}}{\rightarrow}$ are not.

2.2 Comparison of architectures

From our definition of architecture arises a very simple notion of comparison; we define the predicate *weaker* among architectures as follows:

Definition 4 (Weaker)

$$\begin{array}{rcl} A_1 \leq A_2 & \triangleq & ppo_1 \subseteq ppo_2 \wedge \\ & & int_1 \rightarrow int_2 \wedge ext_1 \rightarrow ext_2 \wedge \\ & & ab_1 \subseteq ab_2 \end{array}$$

Theorem 1 (Validity is decreasing)

$$\forall A_1 A_2, A_1 \leq A_2 \Rightarrow$$
$$\forall EX, A_2.valid\ E\ X \rightarrow A_1.valid\ E\ X$$

Proof[in Coq] From $A_1 \leq A_2$, we have $A_1.ghb \subseteq A_2.ghb$, thus if $A_2.ghb$ is acyclic, so is $A_1.ghb$. □

2.2.1 Making validity monotonous

We define here a criterion to check if an execution X running on an architecture A_1 would be valid on a stronger architecture A_2:

$$A_1.check_{A_2} \triangleq acyclic\ \ (A_2.ghb)$$

We show that this criterion characterises an execution running on A_1 that would be valid on A_2:

Theorem 2 (Characterisation)

$$\forall A_1 A_2, A_1 \leq A_2 \Rightarrow$$

$$\forall EX, A_1.valid\ E\ X \wedge A_1.check_{A_2}\ E\ X \leftrightarrow A_2.valid\ E\ X$$

Proof[in Coq]

- $\Rightarrow$ X being valid on A_1, we have all requirements – well formedness and uniproc – to guarantee it is valid on A_2, except the last predicate, which holds by the hypothesis $check_{A_2}$.
- $\Leftarrow$ X being valid on A_2 gives us all requirements – well formedness and uniproc – to guarantee its validity on A_1 except the last one. As $A_1 \leq A_2$, we know that $A_1.ghb \subseteq A_2.ghb$ (lemma ghb_incl), thus the acyclicity requirement for $A_1.ghb$ holds if $A_2.ghb$ is acyclic. □

2.2.2 Examples

Sc In the context of our generic framework, we designed a criterion to decide if a particular execution X, with respect to an event structure E and on an architecture A, is Sc :

$$A.check_{Sc} \triangleq acyclic\ (\overset{\text{po}}{\rightarrow} \cup \overset{\text{hb}}{\rightarrow})$$

This criterion characterises valid weak executions that are Sc :

Corollary 1 (Sc characterisation)

$$\forall AEX, A \leq Sc, A.valid\ E\ X \wedge\ A.check_{Sc}\ E\ X \leftrightarrow Sc.valid\ E\ X$$

Proof[in Coq]

- $\Rightarrow$ As $\overset{\text{po}}{\rightarrow} \cup \overset{\text{hb}}{\rightarrow} = Sc.ghb$, this is a direct consequence of thm. 2.
- $\Leftarrow$ as $A \leq Sc$, this is a direct consequence of thm. 1. □

This result allows us to see that the outcome `0:r1=0; 1:r2=0` for **i3** (leftmost picture in figure 3) will never show up on a sequentially consistent machine. All other executions depicted in fig. 3 are SC by the same argument.

Tso In the context of our generic framework, we designed a criterion to decide if a particular execution X, with respect to an event structure E and on an architecture A, is Tso ; consider $\overset{\text{hb\_tso}}{\Rightarrow}$ to be $\overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{rfe}}{\rightarrow}$:

$$A.check_{Tso} \triangleq acyclic\ (\overset{\text{ppo\_tso}}{\Rightarrow} \cup \overset{\text{hb\_tso}}{\Rightarrow})$$

This criterion characterises valid weak executions that are Tso :

Corollary 2 (Tso characterisation)

$$\forall AEX, A \leq Tso, A.valid\ E\ X \wedge\ A.check_{Tso}\ E\ X \leftrightarrow Tso.valid\ E\ X$$

Proof[in Coq]

$\Rightarrow$ As $\overset{\text{ppo\_tso}}{\Rightarrow} \cup \overset{\text{hb\_tso}}{\Rightarrow} = Tso.ghb$, this is a direct consequence of thm. 2.

$\Leftarrow$ as $A \leq Tso$, this is a direct consequence of thm. 1. □

This result allows us to conclude that all the outcomes for **i3** specified in fig. 3 may show up on a Tso machine.

2.3 Equivalence with native models

2.3.1 Sc is SC

We show that the SC definition from [17] is equivalent to our definition:

Theorem 3 (Sc is SC)

$$\forall EX, Sc.valid\ E\ X \leftrightarrow \exists \overset{ex}{\rightarrow}, seq\ \overset{ex}{\rightarrow}\ \wedge\ SC.wit\ \overset{ex}{\rightarrow}\ = X$$

Proof[in Coq]

$\Rightarrow$ from X being valid on Sc, we have $acyclic\ (\overset{\text{ghb}}{\rightarrow})$, which means $acyclic\ (\overset{\text{hb}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow})$ on Sc. We know by cor. 1 this condition is necessary and sufficient to obtain an equivalent SC execution.

$\Leftarrow$ from the sequential execution $\overset{\text{ex}}{\rightarrow}$, we produce a $SC.wit$ which is valid on any weaker architecture by thm. 1. □

2.3.2 Tso is TSO

We show that the TSO definition from [1] is equivalent to our definition:

Theorem 4 (Tso is TSO)

$$\forall EX, Tso.valid\ E\ X \leftrightarrow \exists \overset{ex}{\rightarrow}, ptso\ \overset{ex}{\rightarrow}\ \wedge\ TSO.wit\ \overset{ex}{\rightarrow}\ = X$$

Proof[in Coq]

$\Rightarrow$ from X being valid on Tso, we know X satisfies $check_{Tso}$ by cor. 2. $check_{Tso}$ gives us an acyclic relation, therefore a partial order on $\mathfrak{E}$, such that its restriction to $\mathfrak{W}$ is the total order on stores required by TSO. As $Tso.ghb$ includes $R*$ and WW by construction, we have the final requirements to provide an execution valid on TSO.

$\Leftarrow$ from $\overset{\text{ex}}{\rightarrow}$, we produce a $TSO.wit$ which is valid on any architecture weaker than Tso by thm. 1. □

2.4 Testing

In this section we precisely define our testing methodology and describe our tools.

2.4.1 Tools

litmus To understand the memory model provided by a given machine M, we use *litmus tests*, which are assembly programs, with specified initial state of memory and registers. To run them on a machine, we use our litmus tool, which runs a C skeleton into which the litmus test is encapsulated. For a given test $\mathbf{t}$ running on M, we collect the final content of memory and registers, thus defining a set of *observed* outcomes $\mathcal{O}_M(\mathbf{t})$.

memevents To compare the memory model as observed on a machine and our theoritecal one, we implemented our generic framework in the memevents tool, written in OCaml. The main module axiom is an implementation of the theory presented at section 2: provided an architecture module A such as $Sc.Arch$ or $Tso.Arch$, it outputs all possible execution witnesses (in the absence of loops) that are valid in the memory model W induced by A – in the sense of the *valid* predicate defined at section 2.1.4, which $final$ define the set of *valid* outcomes $\mathcal{V}_W(\mathbf{t})$. When there are loops, it unfolds them several times, which gives a subset of valid executions, which has been enough for our purposes. Moreover, memevents is able to output a counter example: when a particular outcome is specified, it shows which cycles in the $\overset{\text{ghb}}{\rightarrow}$ relation invalidate this execution. This gives an insight on why this execution is not allowed on a particular architecture, and if barriers are needed or not.

2.4.2 Comparison of models

An additional tool, compare, examines, for a given test $\mathbf{t}$ run on a machine M, the following cases: $\mathcal{O}_M(\mathbf{t}) \subseteq \mathcal{V}_W(\mathbf{t})$, from which we know our model is not invalidated, and $\mathcal{O}_M(\mathbf{t}) \not\subseteq \mathcal{V}_W(\mathbf{t})$, from which we know our model is invalidated. When $\mathcal{O}_M(\mathbf{t}) \subseteq \mathcal{V}_W(\mathbf{t})$, the most challenging case is when $\mathbf{t}$ is in $\mathcal{V}_W(\mathbf{t})$ yet not in $\mathcal{O}_M(\mathbf{t})$, that is it has an outcome which is valid yet not observed. Several reasons explain this situation: either $\mathbf{t}$ has not been run enough to observe it, or the tested machine does not implement the feature highlighted by the test. In that case the model is too permissive with respect to this machine. However, we do not seek the adequation of $\mathcal{O}_M(\mathbf{t})$ and $\mathcal{V}_W(\mathbf{t})$: doing so would lead us to particularise our model so that it renders the model of the tested machine. As we want to give a model of an architecture, we should on the contrary define a looser model which includes the observed outcomes of any machine that implements the architecture.

To be more precise, given an architecture A, a model $W = Wmm(A)$ and an implementation M of A, we define two requirements that must satisfy W to be *valid* and *accurate* with respect to M:

Definition 5 (Validity and accuracy of a model)

$$valid\ W \triangleq \forall M, \forall \boldsymbol{t}, \mathcal{O}_M(\boldsymbol{t}) \subseteq \mathcal{V}_W(\boldsymbol{t})$$

$$accurate_M\ W \triangleq \forall \boldsymbol{t}, \mathcal{V}_W(\boldsymbol{t}) \subseteq \mathcal{O}_M(\boldsymbol{t})$$

	observed	*never* observed
i_5	int = false	int = true
i_6	ext = false	ext = true
i_3	WR $\not\subseteq$ ppo	WR $\subseteq$ ppo
i_4	WW $\not\subseteq$ ppo	WW $\subseteq$ ppo
i_1	RW $\not\subseteq$ ppo	RW $\subseteq$ ppo
i_2	RR $\not\subseteq$ ppo	RR $\subseteq$ ppo

Figure 5: Summary of characteristic tests

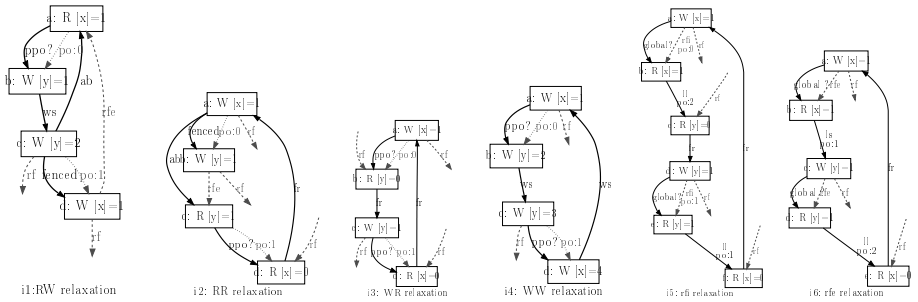


Figure 6: Characteristic tests exhibiting relaxations

Thus we require our model to be *valid* but not necessarily *accurate* with respect to all its implementations.

2.4.3 Characteristic tests

We present here the key tests to understand how to instanciate the parameters of our model. Let us assume given a machine M that implements a model W. The main idea is to observe one relaxation – of the store atomicity or ordering constraints – at the time, by considering an execution where all relations involved are global, except the one in question. Thus, if the specified outcome is observed, M exhibits this relaxation, otherwise there would have been a cycle in the validity check of this execution, which would have been forbidden.

We assume here it is always possible to maintain a $R*$ pair in program order, using a dependency between the two accesses. This does not mean we consider *all* $R*$ pairs to be preserved in program order, but only the ones that have a dependency between them. Maintained RR (resp. RW) pairs will be depicted by ll (resp. ls).

Globality of rfmaps

Internal rfmaps

P_0	P_1
(a) `x` $\leftarrow$ `1`	(d) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `x`	(e) `r3` $\leftarrow$ `y`
`ll`	`ls`
(c) `r2` $\leftarrow$ `y`	(f) `r4` $\leftarrow$ `x`
i5 $r1 = r3 = 1 \wedge r2 = r4 = 0$?	

The test i_5 can be found in [6], with number 2.4: it is claimed to highlight a feature called *intra-processor forwarding*, and illustrates the visibility of *store buffering* to the programmer. If the specified outcome of this test is observed, then we consider $\overset{\text{rfi}}{\rightarrow}$ not to be global on M. If the specified outcome *never* shows up, then $\overset{\text{rfi}}{\rightarrow}$ can be considered global. Indeed, as depicted at fig. 6, if internal $\overset{\text{rf}}{\rightarrow}$ were global, there would be a bold $\overset{\text{rf}}{\rightarrow}$ between events a and $b - W[x]$ and $R[x]$ from P_0 – and between events d and $e - W[y]$ and $R[y]$ from P_1. This would lead to a cycle $a \overset{\text{rfi}}{\rightarrow} b \overset{\text{ppo}}{\rightarrow} c \overset{\text{fr}}{\rightarrow} d \overset{\text{rfi}}{\rightarrow} e \overset{\text{ppo}}{\rightarrow} f \overset{\text{fr}}{\rightarrow} a$ in this execution, which would therefore be forbidden.

External rfmaps

P_0	P_1	P_2
(a) `x` $\leftarrow$ `1`	(b) `r1` $\leftarrow$ `x`	(d) `r2` $\leftarrow$ `y`
	`ls`	`ll`
	(c) `y` $\leftarrow$ `1`	(e) `r3` $\leftarrow$ `x`
i6 $r1 = 1 \wedge r2 = 1 \wedge r3 = 0$?		

The test i_6 also can be found in [6], with number 2.6, or in the litterature under the name WRC [13], and finally in the Power documentation with name $isa1$ [3]. If the specified outcome of this test is observed, then we consider $\overset{\text{rfe}}{\rightarrow}$ not to be global on M. If the specified outcome *never* shows up, then $\overset{\text{rfe}}{\rightarrow}$ can be considered global. Indeed, as depicted at fig. 6, if external $\overset{\text{rf}}{\rightarrow}$ were global, there would be a bold $\overset{\text{rf}}{\rightarrow}$ between events a and $b - W[x]$ from P_0 and $R[x]$ from P_1 – and between events c and $d - W[y]$ from P_1 and $R[y]$ from P_2. This would lead to a cycle $a \overset{\text{rfe}}{\rightarrow} b \overset{\text{ppo}}{\rightarrow} c \overset{\text{rfe}}{\rightarrow} d \overset{\text{ppo}}{\rightarrow} e \overset{\text{fr}}{\rightarrow} a$ in this execution, which would therefore be forbidden.

While observing any $\overset{\text{rf}}{\rightarrow}$ not to be global on a machine, as this is the weakest condition on $\overset{\text{rf}}{\rightarrow}$, one should assume that any machine that implements W has the corresponding parameter to $false$, otherwise W could be invalidated.

Preserved program order

WR **pairs** The test i_3, which is depicted at fig. 1, can be found in [6], with number 2.3a. If the specified outcome is observed, then WR pairs are not preserved in program order: the *ppo* parameter of this machine should not include WR pairs. If the specified outcome *never* shows up, then the *ppo* of this machine includes WR. Indeed, as depicted at fig. 6, if WR pairs were global, there would be a bold $\overset{\text{ppo}}{\rightarrow}$ between events a and $b - W[x]$ and $R[y]$ on P_0 – and between c and $d - W[y]$ and $R[x]$ on P_1. This would lead to a cycle $a \overset{\text{ppo}}{\rightarrow} b \overset{\text{fr}}{\rightarrow} c \overset{\text{ppo}}{\rightarrow} d \overset{\text{fr}}{\rightarrow} a$ in this execution, which would therefore be forbidden.

WW **pairs**

P_0	P_1
(a) x ← 1	(c) y ← 3
(b) y ← 2	(d) x ← 4
i4 $x = 1 \wedge y = 3$?	

If the specified outcome of the test i_4 is observed, then WW pairs are not maintained in program order. If it *never* shows up, then the *ppo* of this machine includes WW. Indeed, as depicted at fig. 6, if WW pairs were global, there would be a bold $\overset{\text{ppo}}{\rightarrow}$ between events a and $b - W[x]$ and $W[y]$ on P_0 – and between c and $d - W[y]$ and $W[x]$ on P_1. This would lead to a cycle $a \overset{\text{ppo}}{\rightarrow} b \overset{\text{ws}}{\rightarrow} c \overset{\text{ppo}}{\rightarrow} d \overset{\text{ws}}{\rightarrow} a$ in this execution, which would therefore be forbidden.

RW **pairs**

P_0	P_1
(a) r1 ← x	(c) y ← 2
(b) y ← 1	fence
	(d) x ← 1
i1 $r1 = 1 \wedge y = 2$?	

If the specified outcome of the test i_1 is observed, then RW pairs are not maintained in program order. If it *never* shows up, then the *ppo* of this machine includes RW. Indeed, as depicted at fig. 6, if RW pairs were global, there would be a bold $\overset{\text{ppo}}{\rightarrow}$ between events a and $b - R[x]$ and $W[y]$ on P_0. If the barrier on P_1 is B-cumulative, it orders events c and d but also c and a. This would lead to a cycle $a \overset{\text{ppo}}{\rightarrow} b \overset{\text{ws}}{\rightarrow} c \overset{\text{ab}}{\rightarrow} a$ in this execution, which would therefore be forbidden.

RR **pairs**

P_0	P_1
(a) x ← 1	(c) r3 ← y
fence	
(b) y ← 1	(d) r4 ← x
i2 $r3 = 1 \wedge r4 = 0$?	

The test i_2 can be found in [6], with number 2.1. If M has global external rf, and preserves WW pairs, and the specified outcome is observed, then RR pairs are not preserved in *ppo*. If the specified outcome is *never* observed under the same hypothesis, then *ppo* includes RR pairs.

If M does not have global external rf, we can consider the same test modified so that a B-cumulative barrier is between the instructions on P_0: the B-cumulativity enforces a global ordering between $(a)W[x]$ on P_0 and $(c)R[y]$ on P_1. If the specified outcome is observed, we can conclude that RR pairs are not in *ppo*: otherwise, there would be a bold $\overset{\text{ppo}}{\rightarrow}$ between c and $d - R[y]$ and $R[x]$ on P_1 – which would lead to a cycle $a \overset{\text{ab}}{\rightarrow} c \overset{\text{ppo}}{\rightarrow} d \overset{\text{fr}}{\rightarrow} a$ in the execution, therefore forbidden.

3 Semantics of barriers

3.1 Barriers guarantee

Let us consider two architectures $A_1 \leq A_2$. We examine here what the barriers provided by A_1 should guarantee to restore A_2.

We note $\overset{\mathrm{rf}_2\setminus^1}{\rightarrow}$ for $\overset{\mathrm{rf}_2{}^?}{\rightarrow} \setminus \overset{\mathrm{rf}_1{}^?}{\rightarrow}$. We define the predicate $A_1.fb$ – for *fully barriered* – on A_1 as follows:

$$\overset{\mathrm{ab}_1}{\rightarrow} = (\overset{\mathrm{rf}_2\setminus^1}{\rightarrow})?; \overset{\mathrm{ppo}_2}{\rightarrow}; (\overset{\mathrm{rf}_2\setminus^1}{\rightarrow})?$$

We show that this a sufficient condition on the barriers provided by A_1 to restore an execution valid on A_2:

Theorem 5 (Barriers guarantee)

$$\forall A_1 A_2, A_1 \leq A_2 \Rightarrow$$

$$\forall EX, A_1.valid\ E\ X \wedge A_1.fb\ E\ X \Rightarrow A_2.valid\ E\ X$$

Proof[in Coq] Suppose that it is not valid on A_2: thus we have a cycle in $A_2.ghb$, that is in $\overset{\mathrm{ws}}{\rightarrow} \cup \overset{\mathrm{fr}}{\rightarrow} \cup \overset{\mathrm{rf}_2{}^?}{\rightarrow} \cup \overset{\mathrm{ppo}_2}{\rightarrow}$. Such a cycle is a cycle in $\overset{\mathrm{ws}}{\rightarrow} \cup \overset{\mathrm{fr}}{\rightarrow} \cup \overset{\mathrm{rf}_1{}^?}{\rightarrow} \cup \overset{\mathrm{ppo}_1}{\rightarrow} \cup (\overset{\mathrm{rf}_2\setminus^1}{\rightarrow})? \cup \overset{\mathrm{ppo}_2}{\rightarrow} \cup (\overset{\mathrm{rf}_2\setminus^1}{\rightarrow})?$ which implies a cycle in $\overset{\mathrm{ws}}{\rightarrow} \cup \overset{\mathrm{fr}}{\rightarrow} \cup \overset{\mathrm{rf}_1{}^?}{\rightarrow} \cup \overset{\mathrm{ppo}_1}{\rightarrow} \cup \overset{\mathrm{ab}_1}{\rightarrow}$ that is, $A_1.ghb$. Thus we contradict the validity of X on A_1. □

This result provides an insight on what power should have a barrier provided by an architecture A_1 – e.g. PowerPC – to restore a stronger model – e.g. SC. First, the barrier should restore the pairs that are preserved in program order on A_2 but not on A_1. Second, the barrier should compensate the lack of relations between writes and reads events – which we model by $\overset{\mathrm{rf}}{\rightarrow}$ not being a global relation in the general case. Thus, if the $\overset{\mathrm{rf}}{\rightarrow}$ relation is not global on A_1 but global on A_2, we overcome the lack of globality of $\overset{\mathrm{rf}}{\rightarrow}$ by ordering the beginning with the end of the chain. This is how we interpret the *cumulativity* of barriers as stated in the PowerPC documentation [3]. We interpret furthermore the *A-cumulativity* (resp. *B-cumulativity*) property, as applying to barriers that enforce ordering of pairs in $\overset{\mathrm{rf}}{\rightarrow}; \overset{\mathrm{po}}{\rightarrow}$ (resp. $\overset{\mathrm{po}}{\rightarrow}; \overset{\mathrm{rf}}{\rightarrow}$). We consider a barrier that only preserves pairs in $\overset{\mathrm{po}}{\rightarrow}$ to be non cumulative.

Provided a barrier that has such power, we also have an insight on where to place these barriers in the code: the statement of the theorem indicates indeed that barriers should be in between any pairs in $\overset{\mathrm{ppo}_2}{\rightarrow}$ such that one of the component of this pair (or both) may give rise to a $\overset{\mathrm{rf}}{\rightarrow}$ relation that is to be global on the stronger architecture A_2 but is not on A_1.

From any architecture to Sc We designed semantics for a barrier that, for any weak memory model induced by an architecture A, would suffice to restablish Sc.

$$e_1 \overset{\mathrm{fenced}}{\rightarrow} e_2 \triangleq \exists b, e_1 \overset{\mathrm{po}}{\rightarrow} b \wedge b \overset{\mathrm{po}}{\rightarrow} e_2$$

$$\overset{\text{fenced}}{\rightarrow} = \overset{\text{po}}{\rightarrow} \qquad \text{(placement)}$$

$$e_1 \overset{\text{ab}}{\rightarrow} e_2 \triangleq e_1 \overset{\text{fenced}}{\rightarrow} e_2 \qquad \text{(base)}$$

$$\vee\, e_1 \overset{\text{rf}}{\rightarrow} r \wedge r \overset{\text{ab}}{\rightarrow} e_2 \qquad \text{(A-cumulativity)}$$

$$\vee\, e_1 \overset{\text{ab}}{\rightarrow} w \wedge w \overset{\text{rf}}{\rightarrow} e_2 \qquad \text{(B-cumulativity)}$$

This barrier orders all pairs in $\overset{\text{po}}{\rightarrow}$ as indicates the base case; it also compensates the eventual lack of visibility of $\overset{\text{rf}}{\rightarrow}$ on A by ordering the two ends of a chain $\overset{\text{rf}}{\rightarrow};\overset{\text{po}}{\rightarrow}$ (resp. $\overset{\text{po}}{\rightarrow}$; $\overset{\text{rf}}{\rightarrow}$) as indicates the A-cumulativity (resp. B-cumulativity) case.

3.2 Considering a weaker guarantee

We said the barrier should at first restore the pairs that are preserved in program order on the stronger architecture. Thus, for the simple case where none of the component of a pair in $\overset{\text{ppo}}{\rightarrow}$ gives rise to a $\overset{\text{rf}}{\rightarrow}$ relation that is global on A_2 but not on A_1, there is no need for a barrier as powerful as above: a barrier that only orders the events that surround it statically would be enough. Consider the wfb predicate: $wfb \triangleq \overset{\text{ppo}_2}{\rightarrow} \setminus \overset{\text{ppo}_1}{\rightarrow}$, the following result arises as a natural corollary of thm. 5.

Corollary 3 (Non cumulative barriers guarantee)

$$\forall A_1 A_2, A_1 \leq A_2 \wedge \overset{rfe_1\ ?}{\rightarrow} = \overset{rfe_2\ ?}{\rightarrow} \Rightarrow$$

$$\forall EX, A_1.valid\ E\ X \wedge A_1.wfb\ E\ X \Rightarrow A_2.valid\ E\ X$$

From Tso to Sc As $\overset{\text{rfe}}{\rightarrow}$ are considered global in both Tso and Sc, we only need a non cumulative barrier to restore Sc from Tso. We define the pairs of writes and reads in $\overset{\text{po}}{\rightarrow}$ as follows: $WR \triangleq \{(w,r) \mid w \overset{\text{po}}{\rightarrow} r\}$. To restore Sc from Tso, we need to preserve the WR pairs, as they are preserved on Sc but not on Tso. As internal $\overset{\text{rf}}{\rightarrow}$ are WR pairs, such a barrier would compensate the lack of visibility of internal $\overset{\text{rf}}{\rightarrow}$ on Tso as well. Therefore we define the following barrier semantics:

$$\overset{\text{fenced}}{\rightarrow} = WR \qquad \text{(placement)}$$

$$e_1 \overset{\text{ab}}{\rightarrow} e_2 \triangleq e_1 \overset{\text{fenced}}{\rightarrow} e_2 \qquad \text{(base)}$$

We define the predicate $Tso.wfb$ as follows: $Tso.wfb \triangleq Tso.ab = WR$, and the following theorem arises as a natural consequence of cor. 3:

Theorem 6 (Barriers placement on Tso)

$$\forall EX, Tso.valid\ E\ X \wedge Tso.wfb\ E\ X \Rightarrow Sc.valid\ E\ X$$

PROOF[in Coq] From X being wfb on Tso, we have $Tso.ghb = \overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{ppo\_tso}}{\Rightarrow} \cup \overset{\text{rfe}}{\rightarrow} \cup WR$, which is acyclic since X is valid on Tso. As WR covers both $\overset{\text{rfi}}{\rightarrow}$ and WR pairs that are not in $\overset{\text{rf}}{\rightarrow}$, we get the acyclicity of $Sc.ghb$ directly. □

From any architecture $A \leq Tso$ to Tso We designed semantics for a barrier that, for any weak memory model induced by an architecture A weaker than Tso, would restablish Tso:

$$\overset{\text{fenced}}{\rightarrow} = \overset{\text{ppo\_tso}}{\Rightarrow} \qquad \text{(placement)}$$

$$e_1 \overset{\text{ab}}{\rightarrow} e_2 \triangleq e_1 \overset{\text{fenced}}{\rightarrow} e_2 \qquad \text{(base)}$$

$$\vee\, e_1 \overset{\text{rfe}}{\rightarrow} r \wedge r \overset{\text{ab}}{\rightarrow} e_2 \qquad \text{(A-cumulativity)}$$

$$\vee\, e_1 \overset{\text{ab}}{\rightarrow} w \wedge w \overset{\text{rfe}}{\rightarrow} e_2 \qquad \text{(B-cumulativity)}$$

Here, all pairs except WR are to be preserved in program order, which is depicted by the placement condition $\overset{\text{fenced}}{\rightarrow} = \overset{\text{ppo\_tso}}{\Rightarrow}$. As internal $\overset{\text{rf}}{\rightarrow}$ are not considered global in Tso, there is no need to compensate them: the ordering power of the barrier concerns only the external $\overset{\text{rf}}{\rightarrow}$, as depicted by the A- and B-cumulativity case.

We retrieve what is described in the Sparc V9 documentation [2]: TSO is indeed obtained from PSO, which is obtained from RMO by barriers placements. In our framework, we define Rmo and Pso as follows – where $R*_l$ (resp. WW_l) represents all pairs of $R*$ (resp. WW) to the same location:

$$Rmo.Arch \triangleq (R*_l \cup WW_l, false, true, ab1)$$
$$Pso.Arch \triangleq (R* \cup WW_l, false, true, ab2)$$

As for Tso, we deduce from the Val axiom that external $\overset{\text{rf}}{\rightarrow}$ are global, whereas internal are not.

The documentation specifies that PSO is obtained from RMO by adding *LoadLoad* and *LoadStore* barriers after each read. This statement has two consequences in our framework: first, $R*$ pairs are preserved in Pso, and second, to restore Pso from Rmo, since the external $\overset{\text{rf}}{\rightarrow}$ are already global on Rmo, one should use a non cumulative barrier that preserves $R*$ pairs.

TSO is obtained from PSO by adding *StoreStore* barriers after each write. We conclude that WW pairs are preserved in Tso, and that to restore Tso from Pso, one should use a non cumulative barrier that preserves WW pairs. Thus, from Rmo to Tso, a non cumulative barrier that preserves $R*$ and WW is needed, as stated by the cor. 3.

4 Case study: a Power model

In this section we define an architecture for Power, that is, we define relations $\overset{\text{ppo}}{\rightarrow}$ and $\overset{\text{ab}}{\rightarrow}$ and booleans *int* and *ext*. We do not claim for a definitive PowerPC

model; we rather confront a tentative PowerPC model against actual PowerPC machines.

4.1 Complete event structures and execution witnesses

Henceforth, we will reason on complete event structures, on which section 2 abstract. We shall avoid exhaustive treatment of complete event structures, interested readers may refer to [18, 9]; instead, we sketch the main ideas.

Additional events In addition to memory events, the execution of an instruction may generate a variety of events: most instructions generate *register events* that render accesses to registers, memory barriers instructions generate *barrier events* and conditional branch instructions generate *commit events* that express branching decisions. We note $\mathfrak{B}$ the set of barrier events and $\mathfrak{C}$ the set of commits, b and c being typical elements. We shall handle three memory barrier instructions : `isync`, `sync` and `lwsync`. The corresponding events are distinguished by predicates is-isync, etc. As in previous sections, we still denote the set of memory events by $\mathfrak{E}$ (typical element e), the set of memory read events by $\mathfrak{R}$ (typical element r), and the set of memory write events by $\mathfrak{W}$ (typical element w).

Extended or additional relations We extend the program order relation $\overset{\text{po}}{\rightarrow}$ to all events. In particular, $\overset{\text{po}}{\rightarrow}$ now orders both memory and barrier events. Moreover, complete event structures comprise additional relations, more specifically *intra instruction causality* $\overset{\text{iico}}{\rightarrow}$ that represents the ordering constraints of events within a same instruction. Moreover, the following relation $\overset{\text{fenced(k)}}{\longrightarrow}$ renders the presence of a barrier of style k between memory events e_1 and e_2:

$$e_1 \overset{\text{fenced}(k)}{\longrightarrow} \triangleq \exists b, \text{is-}k(b) \wedge e_1 \overset{\text{po}}{\rightarrow} b \overset{\text{po}}{\rightarrow} e_2.$$

Execution witnesses also become more complete, as relation $\overset{\text{rf}}{\rightarrow}$ now relates register events. We note $\overset{\text{rf-reg}}{\rightarrow}$ the subrelation of $\overset{\text{rf}}{\rightarrow}$ that relates register stores to register loads that read their values. As a side note, relation $\overset{\text{rf-reg}}{\rightarrow}$ derives from sequential execution in a much stronger sense than $\overset{\text{rf}}{\rightarrow}$ on memory. Namely, $w \overset{\text{rf-reg}}{\rightarrow} r$ when w is maximal amongst the predecessors of r in program order.

Illustration Figure 7 shows a program fragment and a fragment of the corresponding complete event structure, together with an execution witness.

The first instruction is an indexed load from memory `lwz`: base address y is taken from register `r5` (which has been writen into elsewhere), index is 0, and the value read from the effective address $y + 0$ is stored into register `r2`. We here view three events, labelled on figure as c, a and d. As an example of intra-instruction dependency, $a \overset{\text{iico}}{\rightarrow} d$ expresses that loading from memory precedes storing into register `r2`.

The compare and branch sequence is less trivial: the instruction `cmpwi r2,1` compares the content of `r2` to the constant `1` and stores the result of comparison (2 means equality) into the control register `cr0`. The next instruction is a conditional branch `bne`, with branching decision (commit event h) conditioned by the contents of control register `cr0` ($g \overset{\text{iico}}{\rightarrow} h$). Instruction `bne` is "branch not

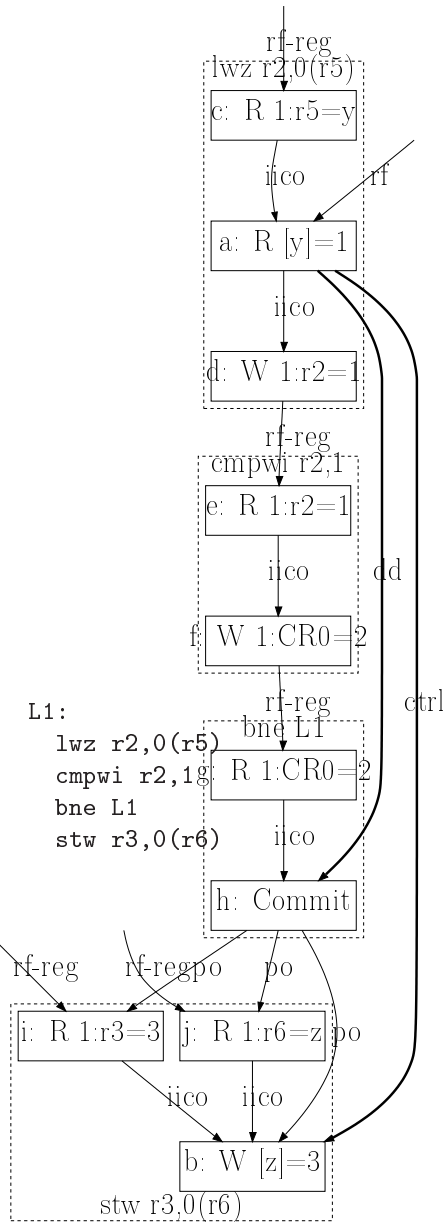


Figure 7: Example of preserved program order

equal". Thus, since control registers signals equality, the branch is not taken and the next instruction to be executed is the store `stw`, as shown by the arrows $h \overset{\text{po}}{\rightarrow} i$, $h \overset{\text{po}}{\rightarrow} j$ and $h \overset{\text{po}}{\rightarrow} b$.

4.2 Globality of rfmaps

Running tests i_5 and i_6 on a Power machine yields the specified outcomes. Thus, we consider the $\overset{\text{rfi}}{\rightarrow}$ and $\overset{\text{rfe}}{\rightarrow}$ relations not to be global for Power.

4.3 Preserved program order *ppo*

Some parts of the $\overset{\text{po}}{\rightarrow}$ program order relation are reflected in the global happens-before relation. In this section we pick those out, defining a preserved-program-order relation $\overset{\text{ppo}}{\rightarrow}$.

Data dependencies Data dependencies within a processor arise from any combination of the reads-from relation on registers and the intra-instruction causality relation:

$$\overset{\text{dd}}{\rightarrow} \quad \triangleq \quad (\overset{\text{rf-reg}}{\rightarrow} \cup \overset{\text{iico}}{\rightarrow})^+$$

(here R^+ denotes the transitive closure of R). Note that this relation includes *no* dependencies via memory.

The restriction of the above to memory events is written $\overset{\text{dd-mem}}{\rightarrow}$:

$$\overset{\text{dd-mem}}{\rightarrow} \quad \triangleq \quad \overset{\text{dd}}{\rightarrow} \cap (\mathfrak{M} \times \mathfrak{M})$$

Control dependencies A memory write is *control dependent* on a memory read if there is an intervening commit (of a conditional branch) that is data-dependent on the read and precedes (in program order) the write:

$$r \overset{\text{ctrl}}{\rightarrow} w \quad \triangleq \quad \exists c \in \mathfrak{C}.\ r \overset{\text{dd}}{\rightarrow} c \overset{\text{po}}{\rightarrow} w$$

This relation models that fact that memory writes are not speculated, whereas reads can be.

Isync dependencies A memory event is isync-dependent on a memory read if there exists an intervening commit (of a conditional branch) that is data-dependent on the read and is separated (in program order) by an isync from the event:

$$r \overset{\text{isync}}{\rightarrow} m \quad \triangleq \quad \exists c \in \mathfrak{C}.\ r \overset{\text{dd}}{\rightarrow} c \wedge c \overset{\text{fenced(isync)}}{\rightarrow} m$$

Note that this only adds anything beyond control dependencies in the case of a memory read/read pair.

Figure 7 gives an example of a control dependency, from load a to store b through commit h.

All together The preserved program order relation is just the union of data dependency for memory events, control dependencies, and isync dependencies:

$$\overset{\text{ppo}}{\rightarrow} \triangleq \overset{\text{dd-mem}}{\rightarrow} \cup \overset{\text{ctrl}}{\rightarrow} \cup \overset{\text{isync}}{\rightarrow}$$

Note that memory stores are never the source of a $\overset{\text{ppo}}{\rightarrow}$ pair, as a consequence of the instruction semantics: a memory store cannot be the source of a $\overset{\text{iico}}{\rightarrow}$ pair, nor of a $\overset{\text{rf-reg}}{\rightarrow}$ pair. Thus, a natural partition of $\overset{\text{ppo}}{\rightarrow}$ pairs is into load/load pairs and load/store pairs. We include these pairs in the global happens-before relation:

- for load-load pairs: we refer to [5, pp. 653–668], which states that in such a situation, load r_1 will be performed before load r_2 with respect to any processor, which we interpret as r_1 being globally performed before r_2. It is not clear to us whether or not these notions are equivalent in the case of a load, or if our interpretation makes sense w.r.t architectural insights;
- for load-store pairs: we deduce from $r \overset{\text{ppo}}{\rightarrow} w$ that r happens before the store is initiated. Since we consider loads to be atomic – thus considering they are globally performed as soon as they are initiated – and stores to be initiated before being globally performed, we deduce that r is globally performed before w is, and include such $\overset{\text{ppo}}{\rightarrow}$ pairs in the global happens-before relation.

4.4 Values do not come out of thin air

In the application of our framework for Power, load-store pairs endure a particular treatment: we extend load-store $\overset{\text{ppo}}{\rightarrow}$ edges by following $\overset{\text{rf}}{\rightarrow}$ ones, thus defining another relation $\overset{\text{ppo-ext}}{\rightarrow}$, which we will also include in the global happens-before relation.

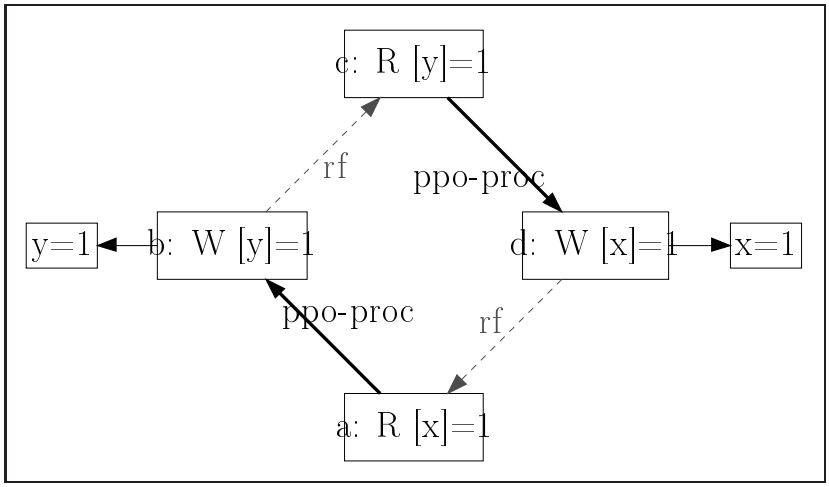
Consider a triple $r \overset{\text{ppo}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} r'$, where r' does not need to originate from the same processor as r and w. Here r happens before the store is initiated, and r' happens after the store is initiated. Thus, intuition suggests that r globally happens before r'. We define the extension of $\overset{\text{ppo}}{\rightarrow}$ by $\overset{\text{rf}}{\rightarrow}$ as follows:

$$r \overset{\text{ppo-ext}}{\rightarrow} r' \triangleq \exists w,\ r \overset{\text{ppo}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} r'$$

Our extension of load-store dependencies is a generalization of some check that weak models often – and arguably must – incorporate: the "causality check" of [10], or the "values do no not come out of thin air" of [7]. The canonical example of such a check is as follows:

P_0	P_1
(a) `r1 ← x`	(c) `r1 ← y`
(b) `y ← r1`	(d) `x ← r1`

We further assume that x and y initially contain 0. Without specific provision in the model, the absurd outcome $x = 1; y = 1$ might remain valid, as demontrated by the following execution witness:



Our model invalidates the execution thanks to ppo extension. Namely, we have $a \overset{\text{ppo-ext}}{\rightarrow} c$ (by $a \overset{\text{ppo-proc}}{\rightarrow} b \overset{\text{rf}}{\rightarrow} c$) and $c \overset{\text{ppo-ext}}{\rightarrow} a$ (by $c \overset{\text{ppo-proc}}{\rightarrow} d \overset{\text{rf}}{\rightarrow} a$. Hence, $\overset{\text{ppo}}{\rightarrow}$ alone is cyclic. *A fortiori* $\overset{\text{ghb}}{\rightarrow}$ is cyclic, since $\overset{\text{ppo}}{\rightarrow}$ is included in $\overset{\text{ghb}}{\rightarrow}$. Note that we could prevent values to come out of thin air by adding another sanity check on the $\overset{\text{rf}}{\rightarrow}$ relation in our generic framework, following Alpha's documentation.

There are two reasons for considering such an extension to be global – that is, included in the global happens before relation:

- it rules out some examples in which values would appear out of thin air as presented in the following example;
- from a global time perspective, r and r' are ordered respectively before and after the point of time when w is initiated; for more details, see section 6 where an example is discussed.

This is perhaps intuitively plausible, and it suffices to rule out some examples in which values would appear out of thin air (we suppose here that the architecture should rule out thin-air reads, though that might be debated), and to correspond with our Power5 experiments on the test presented in section 6. However, such an extension does not seem to be forced. Therefore, it is not clear to us whether we should forbid that behaviour in the model.

Illustration We consider here the litmus test **adir1v3** (a variation on [7, Test 1]).

Figure 8 shows the program and a non-*SC* execution ($\overset{\text{hb}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$ is cyclic). One may first notice the $\overset{\text{ppo}}{\rightarrow}$ relation between load b and store c. It follows from a data dependency, since the effective address of c is exactly the value read by b (*i.e.* the address of location y). The relation $\overset{\text{ghb}}{\rightarrow}$ is highlighted with black bold arrows. We have:

1. $b \overset{\text{ppo}}{\rightarrow} c$ (data dependency), and thus $b \overset{\text{ppo-ext}}{\rightarrow} d$ (by $b \overset{\text{ppo}}{\rightarrow} c \overset{\text{rf}}{\rightarrow} d$ and ppo-extension).
2. $d \overset{\text{sync}}{\rightarrow} e$ (sync instruction), and thus $c \overset{\text{ab}_e}{\rightarrow}$ (by $c \overset{\text{rf}}{\rightarrow} d \overset{\text{sync}}{\rightarrow} e$ and A-cumulativity).

P_0	P_1
(a) `x ← &y`	(d) `r3 ← y`
(b) `r6 ← x`	`sync`
(c) `*r6 ← 1`	(e) `r4 ← x`
Initially: `x=&z;`	
Allowed: `1:r3=1; 1:r4=&z;`	

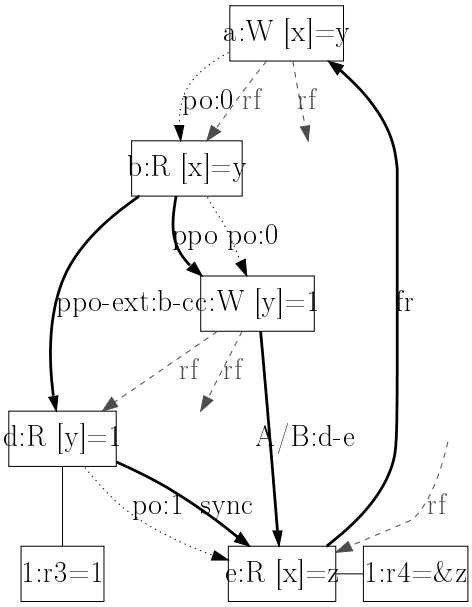


Figure 8: Litmus test **adir1v3**

3. $e \overset{\text{fr}}{\rightarrow} a$, by definition of $\overset{\text{fr}}{\rightarrow}$.

Clearly, $\overset{\text{ghb}}{\rightarrow}$ is acyclic and the execution is valid. In some sense, validity follows from $a \overset{\text{rf}}{\rightarrow} b$ not being global — since there is a cycle $a \overset{\text{rf}}{\rightarrow} b \overset{\text{ghb}}{\rightarrow} a$. Note that even if we strengthen the $\overset{\text{ppo}}{\rightarrow}$ relation by its extension, this outcome is still valid.

This $\overset{\text{rf}}{\rightarrow}$ relation is internal to a processor; by considering it not to be global, we model the presence of a store buffer on this processor, which we suppose to be at least a part of the reason why this behaviour is observed.

4.5 Cumulative memory barriers

`sync` The `sync` barrier is the incarnation of the SC-restoring cumulative barrier described in section 3.1, which definition we expand for clarity:

$$\overset{\text{sync}}{\rightarrow} \triangleq \overset{\text{fenced(sync)}}{\rightarrow}$$

$$\begin{aligned} e_1 \overset{\text{ab-sync}}{\rightarrow} e_2 \triangleq\ & e_1 \overset{\text{sync}}{\rightarrow} e_2 && \text{(base)} \\ \vee\ & e_1 \overset{\text{rf}}{\rightarrow} r \overset{\text{sync}}{\rightarrow} e_2 && \text{(A-cumulativity)} \\ \vee\ & e_1 \overset{\text{sync}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} e_2 && \text{(B-cumulativity)} \\ \vee\ & e_1 \overset{\text{rf}}{\rightarrow} r \overset{\text{sync}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} e_2 && \text{(A/B-cumulativity)} \end{aligned}$$

`lwsync` PowerPC features a lightweight cumulative barrier, `lwsync`, which semantics we define as follows:

$$\overset{\text{lwsync}}{\rightarrow} \triangleq \overset{\text{fenced(lwsync)}}{\rightarrow} \cap ((\mathfrak{W} \times \mathfrak{W}) \cup (\mathfrak{R} \times \mathfrak{E}))$$

$$e_1 \overset{\text{ab-lw}}{\rightarrow} e_2 \triangleq e_1 \overset{\text{lwsync}}{\rightarrow} e_2 \qquad \text{(base)}$$

$$\vee\, e_1 \overset{\text{rf}}{\rightarrow} r \wedge r \overset{\text{ab-lw}}{\rightarrow} e_2 \wedge e_2 \in \mathfrak{W} \qquad \text{(A-cumulativity)}$$

$$\vee\, e_1 \overset{\text{ab-lw}}{\rightarrow} w \wedge w \overset{\text{rf}}{\rightarrow} e_2 \wedge e_1 \in \mathfrak{R} \qquad \text{(B-cumulativity)}$$

In other words, `lwsync` acts as `sync` except on store-load pairs, the exception impacting both the base and cumulativity case.

Finally we define relation $\overset{\text{ab}}{\rightarrow}$ as the union of $\overset{\text{ab-sync}}{\rightarrow}$ and $\overset{\text{ab-lw}}{\rightarrow}$.

Analogy between $\overset{\text{ppo-ext}}{\rightarrow}$ and B-cumulativity The $\overset{\text{ppo-ext}}{\rightarrow}$ extension is arguably an analog of B-cumulativity. Here we conjecture that barriers implement A-cumulativity by waiting for some stores to be performed globally, in which case $\overset{\text{ppo}}{\rightarrow}$ ignores the issue. By contrast, B-cumulativity on a load-store pair demands no specific actions, as the natural consequence of a $w \overset{\text{rf}}{\rightarrow} r$ implying that r is performed only once w is issued.

5 Barrier experiments

5.1 Official tests

A programming note in [3, p. 415] describes two examples in precise prose. We formulate those as invalid executions of litmus tests.

isa1 Let us first examine the simpler test **isa1**, given as pseudo-code at the top of figure 9. PowerPC documentation states: "*Cumulative ordering dictates that the value loaded from location x by processor 2 is 1.*" Our interest is in an officially invalid execution, which our Power model should also deem invalid (bottom of figure). Thus, we interpret the above prescription as forbidding the value loaded from location x by processor 2 to be 0, the initial contents of x.

To relate an execution graph to a litmus test, one may first relate events to instructions, using the event annotations ((a), (b),...) in program text and the $\overset{\text{po}}{\rightarrow}$ arrows in graphs. For instance, P_1 performs a load from location x, reading 1, (event b), a store of 2 to location y (event c), and those are separated by a `sync` instruction (relation $b \overset{\text{sync}}{\rightarrow} c$). Other arrows are as follows: dashed arrows give relation $\overset{\text{rf}}{\rightarrow}$, with pending arrows to read events being loads from intial state, and pending arrows from write events being stores to final state; while bold arrows give relation $\overset{\text{ghb}}{\rightarrow}$. For instance, $e \overset{\text{fr}}{\rightarrow} a$ results from event e reading the initial value of location x, which is overwritten by event a. Or, $a \overset{\text{ab}}{\rightarrow} d$ results from barrier cumulativity by $a \overset{\text{rf}}{\rightarrow} b \overset{\text{sync}}{\rightarrow} c \overset{\text{rf}}{\rightarrow} d$.

We can now reach interesting conclusions quite easily: by cor. 1, the execution shown is not SC, since there is a cycle $a \overset{\text{rf}}{\rightarrow} b \overset{\text{po}}{\rightarrow} c \overset{\text{rf}}{\rightarrow} d \overset{\text{po}}{\rightarrow} e \overset{\text{fr}}{\rightarrow} a$. More important, the execution is not valid in the Power model, since there are cycles in $\overset{\text{ghb}}{\rightarrow}$. It is worth noticing that the test is presented by [3] as illustrating

isa1

P_0	P_1	P_2
(a) `x ← 1`	(b) `r1 ← x`	(d) `r1 ← y`
	`sync`	`sync`
	(c) `y ← 2`	(e) `r2 ← x`

Forbidden: `1:r1=1; 2:r1=2; 2:r2=0;`

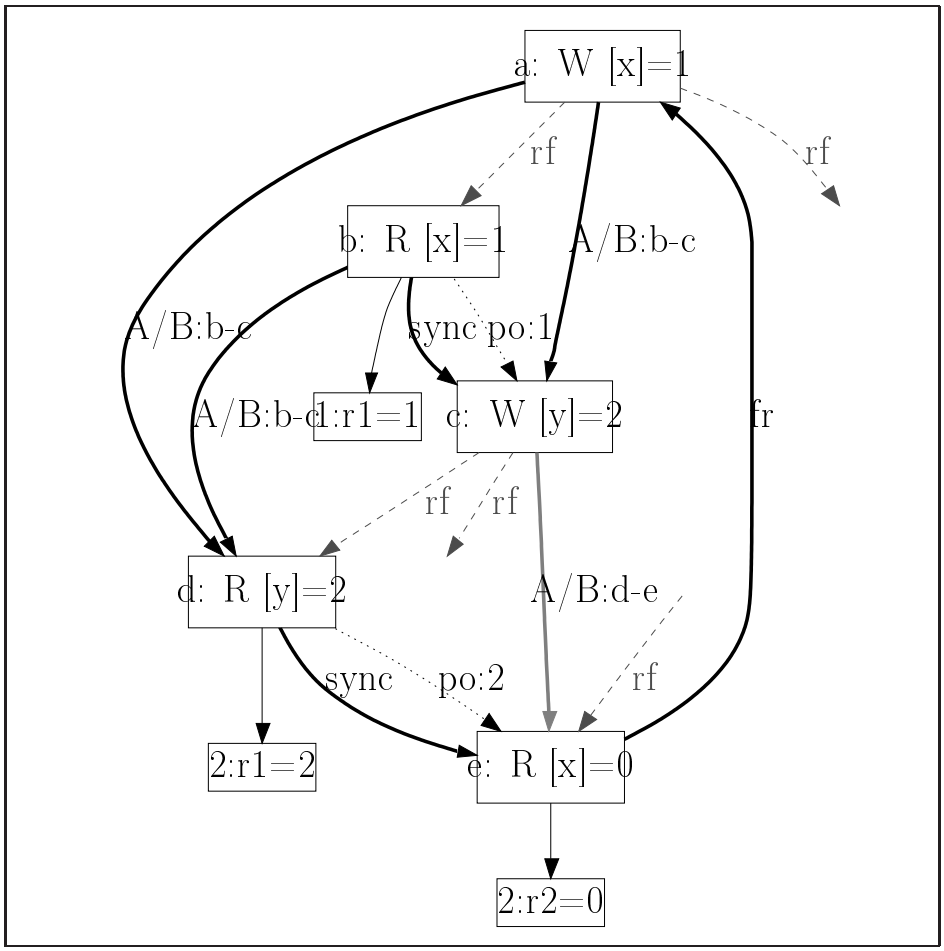


Figure 9: Officialy invalid execution of **isa1**

"*cumulative ordering of storage accesses preceeding a memory barrier*". As a consequence, the cycle $a \overset{\text{ab}}{\rightarrow} c \overset{\text{ab}}{\rightarrow} e \overset{\text{fr}}{\rightarrow} a$ is the most illustrative. Namely, $a \overset{\text{ab}}{\rightarrow} c$ follows from $a \overset{\text{rf}}{\rightarrow} b$ and $b \overset{\text{sync}}{\rightarrow} c$; while $c \overset{\text{ab}}{\rightarrow} e$ follows $c \overset{\text{rf}}{\rightarrow} d$ and $d \overset{\text{sync}}{\rightarrow} e$;

isa2 The more complex test **isa2** (figure 10) is a refinement of **isa1**: a chain of store-reads from P_0 to P_2 that passes through P_1.

But P_0 now performs two stores to locations x and y, separated by a sync; while P_1 loops loading y untils it reads the value 2 written to y by P_0, before storing value 3 to location z. P_2 remains essentially unchanged. As for **isa1**, the

isa2

P_0	P_1	P_2
(a) `x ← 1`	`L1:`	(f) `r3 ← z`
`sync`	(c,d) `r2 ← y`	`sync`
(b) `y ← 2`	`cmp r2,2`	(g) `r1 ← x`
	`bne L1`	
	(e) `z ← 3`	

Forbidden: `2:r3=3; 2:r1=0;`

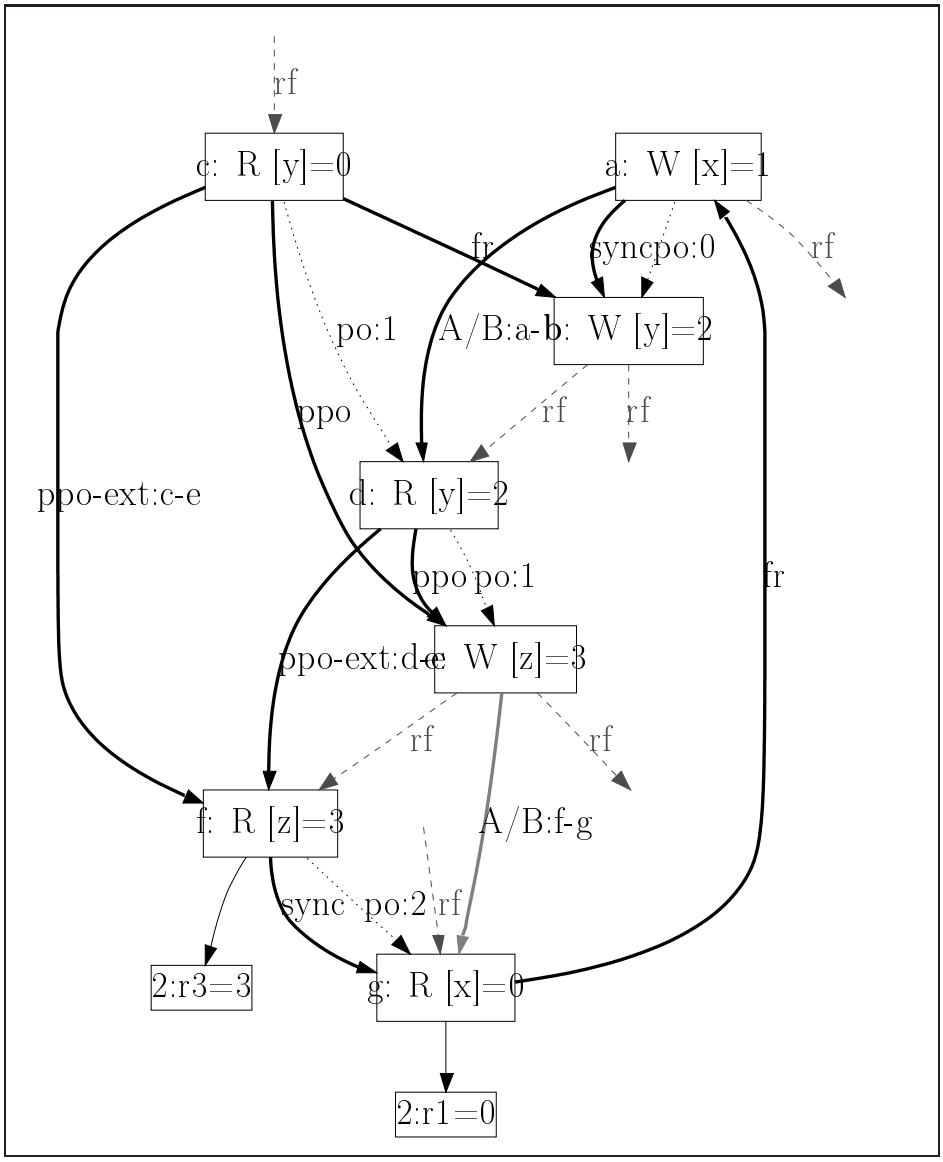


Figure 10: Officially invalid execution of isa2

architecture specification forbids that P_2 loads value 0 from location x (third instruction) when it has loaded (first instruction) the value (here 3) stored by P_1 in some memory location used for communicating (here z). A key observation is the *absence* of a barrier in P_1 code. Instead, we have a control dependency. The example being official, we assume that such a control dependency suffices to prevent the last load of P_2 from reading value 0.

In presence of a conditional branch, there is a clear distinction between program text and execution, or, more precisely between programm listing order and program order $\overset{\text{po}}{\rightarrow}$. We select a particular (invalid) execution witness generated by memevent, where P_1 executes two loop iterations (figure 10). The control dependency is expressed as the two edges $c \overset{\text{ppo}}{\rightarrow} e$ and $d \overset{\text{ppo}}{\rightarrow} e$. The execution is non-SC, by the existence of cycle $a \overset{\text{po}}{\rightarrow} b \overset{\text{rf}}{\rightarrow} d \overset{\text{po}}{\rightarrow} e \overset{\text{rf}}{\rightarrow} f \overset{\text{po}}{\rightarrow} g \overset{\text{fr}}{\rightarrow} a$. The execution is also invalid in our Power model, since $\overset{\text{ghb}}{\rightarrow}$ is cyclic. We clearly identify two cycles: $a \overset{\text{ab}}{\rightarrow} d \overset{\text{ppo}}{\rightarrow} e \overset{\text{ab}}{\rightarrow} g \overset{\text{fr}}{\rightarrow} a$ and $a \overset{\text{ab}}{\rightarrow} d \overset{\text{ppo-ext}}{\rightarrow} f \overset{\text{sync}}{\rightarrow} g \overset{\text{fr}}{\rightarrow} a$. Note that the $\overset{\text{ppo-ext}}{\rightarrow}$ extension is not needed to conclude that this outcome is invalid in our Power model.

5.2 Classical tests

In the previous section, we have demonstrated that our Power model is correct w.r.t. the two official litmus tests that are publicly available. Clearly, two tests are unsufficient to draw any conclusion and we need more.

Some litmus tests are conventional, such as **iriw** (Independant Reads of Independent Writes, figure 11) and **rwc** (Read To Write Causality, figure 12) — see for instance [13], and [4, Example 7.7].

Figures 11 and 12 show non-SC execution witnesses, which are the ones of interest. To see that the executions we consider are non-SC, it suffices to follow $\overset{\text{rf}}{\rightarrow}$, $\overset{\text{fr}}{\rightarrow}$ and $\overset{\text{po}}{\rightarrow}$ arrows in any graph, so as to find a cycle. These graphs also show that the executions considered are invalid our Power model, by the presence of bold $\overset{\text{ghb}}{\rightarrow}$ cycles.

We cannot conclude from the public documentation [5] whether these two tests are invalid on the Power architecture; therefore we resort to experimentation.

5.3 Experiments

In experiments, we observe a selection of the final values of registers and of memory location, yielding outcomes. In the case of the four litmus tests **isa1**–**rwc** the final values of registers written to by the load instructions suffice to identify the non-SC execution depicted. For instance the outcome `[1:r1=1; 1:r2=0; 2:r3=0;]` suffices to identify the non-SC execution of **rwc**.

We performed experiments on two machines `doko` and `hpcx`. `doko` is a 4-cores Power5 machine, running Linux; while `hpcx` is one 16-cores eServer 575, running AIX.

iriw

P_0	P_1	P_2	P_3
(a) `r1` $\leftarrow$ `x`	(c) `r2` $\leftarrow$ `y`	(e) `x` $\leftarrow$ `1`	(f) `y` $\leftarrow$ `2`
`sync`	`sync`		
(b) `r2` $\leftarrow$ `y`	(d) `r1` $\leftarrow$ `x`		

Observed? `0:r1=1; 0:r2=0; 1:r2=2; 1:r1=0;`

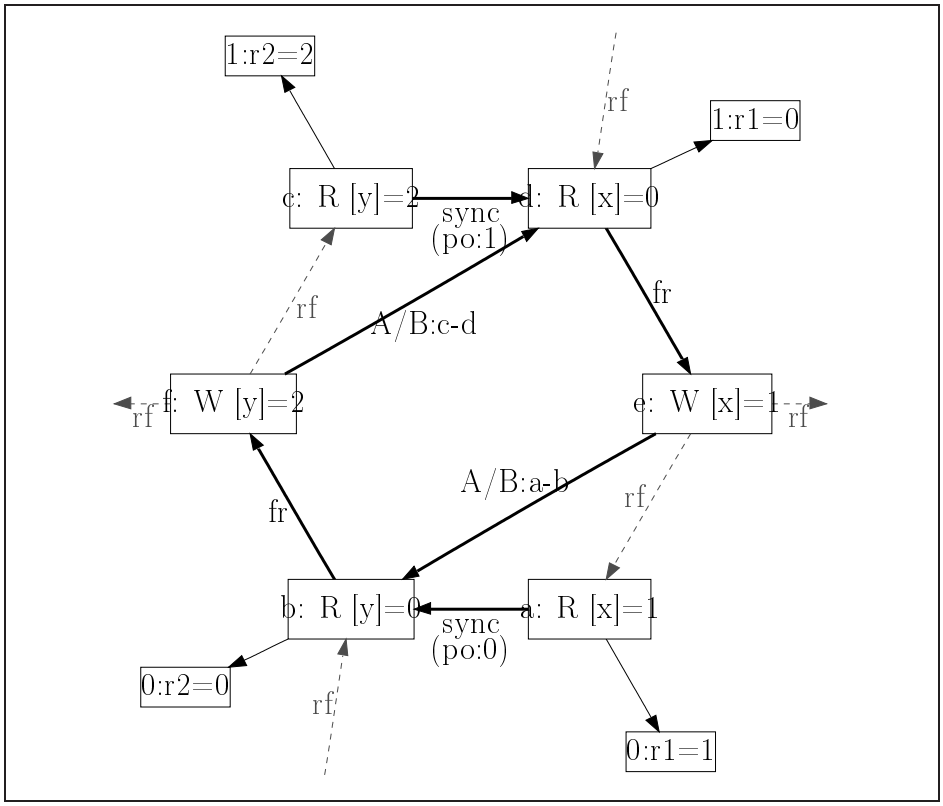


Figure 11: A non-SC execution of litmus test **iriw**

rwc

P_0	P_1	P_2
(a) `x` ← `1`	(b) `r1` ← `x`	(d) `y` ← `1`
	`sync`	`sync`
	(c) `r2` ← `y`	(e) `r3` ← `x`

Observed? `1:r1=1; 1:r2=0; 2:r3=0`

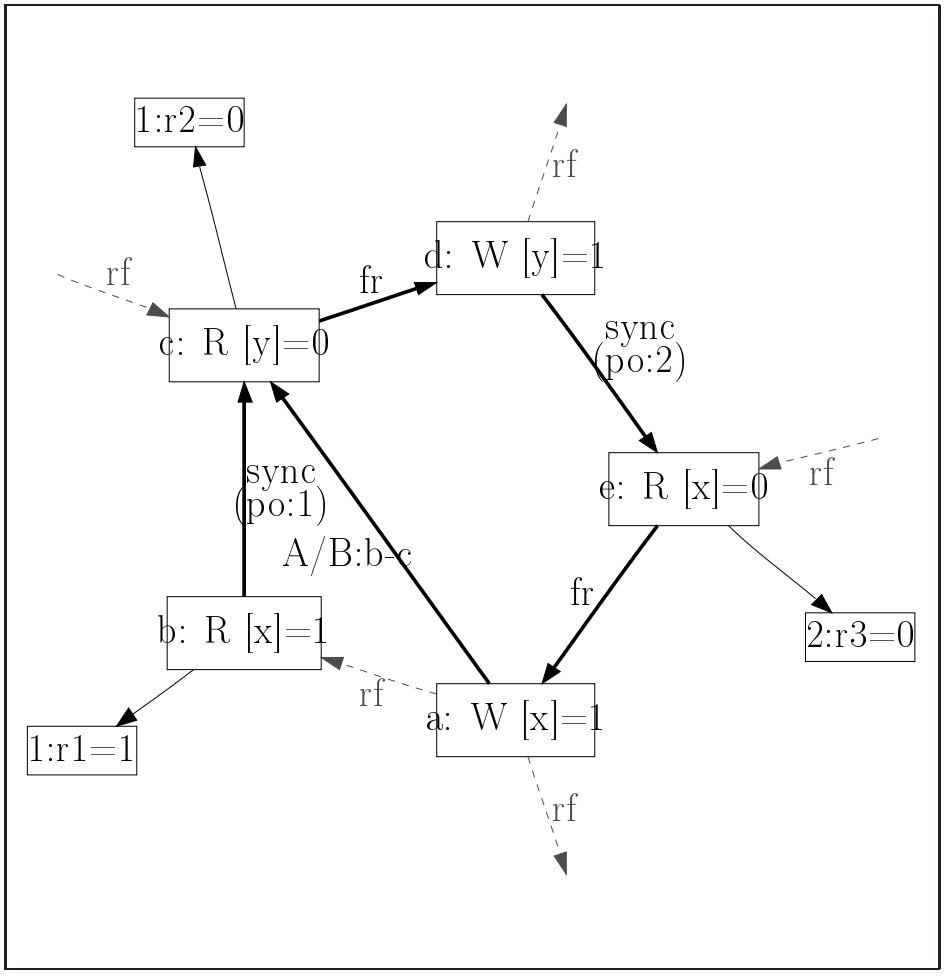


Figure 12: A non-SC execution of litmus test **rwc**

isa2v1

P_0	P_1	P_2
(a) `x` $\leftarrow$ `1`	`L1:`	(f) `r3` $\leftarrow$ `z`
`sync`	(c,d) `r2` $\leftarrow$ `y`	`r4` $\leftarrow$ `xor(r3,r3)`
(b) `y` $\leftarrow$ `2`	`cmp r2,2`	(g) `r1` $\leftarrow$ `*(&x+r4)`
	`bne L1`	
	(e) `z` $\leftarrow$ `3`	

Unobserved: `2:r3=3`; `2:r1=0`;

Figure 13: **isa2v1**

6 Towards a stronger model

We have provided a valid and accurate model for the Power architecture. However, our model may be, to some extent, too weak to program above. We suggest here some extension to make our model stronger yet still valid.

6.1 Extension $\overset{\text{ppo-ext}}{\longrightarrow}$

We have already presented the $\overset{\text{ppo-ext}}{\longrightarrow}$ extension of the $\overset{\text{ppo}}{\longrightarrow}$ relation. However, we did not use it in our preceding reasonings. Let us here consider the **isa2v1**, depicted at fig. 13, which is a variation on **isa2** presented at section 5.1. The execution witness we want to invalidate is depicted at fig. 14.

The code of P_2 changes: a `sync` instruction is replaced by a data dependency from the value of load f to the effective address of load g. As a result, we now have $f \overset{\text{ppo}}{\longrightarrow} g$, where we previously had $f \overset{\text{sync}}{\longrightarrow} g$. Notice that the dependency is a so-called false dependency: the load instruction g always reads location x, regardless of the value read by the load instruction f. Nevertheless, we have $f \overset{\text{ppo}}{\longrightarrow} g$.

We have collected about 750 millions of machine outcomes for this test: outcome `2:r3=3; 2:r1=0` remains unobserved. One corresponding execution (with P_1 loop being executed twice) is as follows:

We see that the pair $d \overset{\text{ppo-ext}}{\longrightarrow} f$ is necessary to invalidate the execution.

We ignore whether the Power architecture would deem the outcome as valid or not; however, experiments suggest that this outcome never shows up. To include such a prescription in our model, we extend $\overset{\text{ppo}}{\longrightarrow}$ into $\overset{\text{ppo-ext}}{\longrightarrow}$.

6.2 Semantics of `lwsync`

We provide a rather weak – that is, permissive – semantics of `lwsync`. However, some programming patterns suggest a stronger semantics for this barrier. Let us indeed consider the following test **lwsync**, depicted at fig. 15, and the associated execution witness of interest, depicted at fig. 16.

W.r.t the semantics of `lwsync` provided at section 4.5, this execution witness – and thus the associated outcome – is valid. However, this use of `lwsync` seems to be common practice amongst low-level programmers, and we have not observed it – yet. As for the $\overset{\text{ppo-ext}}{\longrightarrow}$ extension, we do not know whether this execution should be considered valid w.r.t the Power architecture. However, if so, we would need to extend our framework a bit to handle such a behaviour.

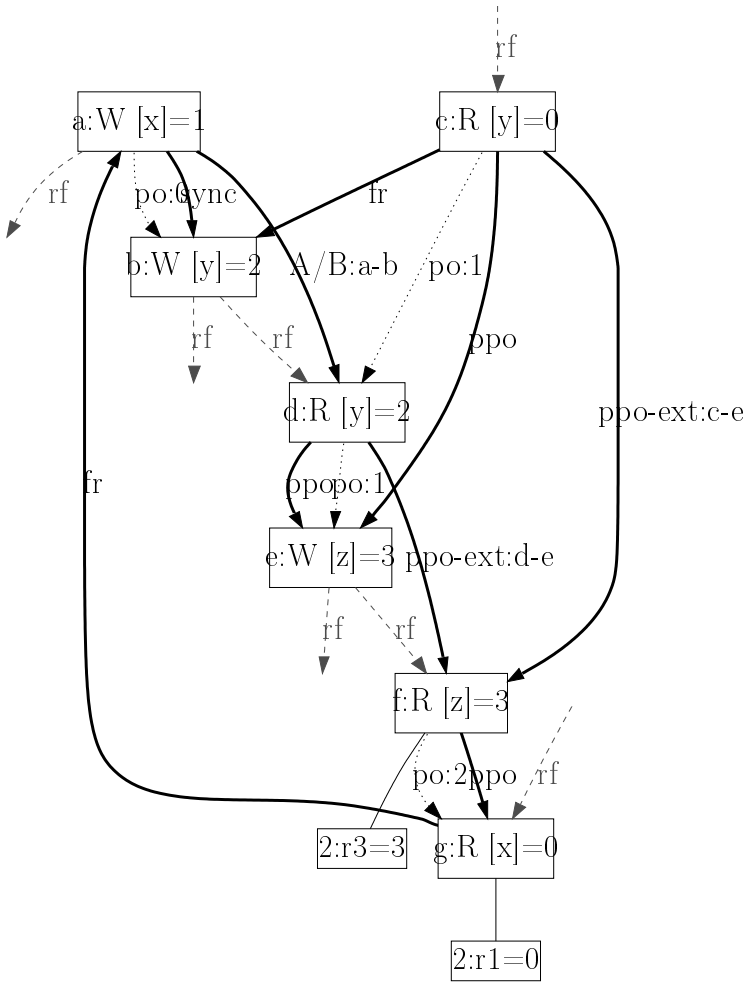


Figure 14: Not observed execution of **isa2v1**

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `r1` $\leftarrow$ `y`
`lwsync`	`ll`
(b) `y` $\leftarrow$ `1`	(d) `r2` $\leftarrow$ `x`
lwsync $\quad x = 1 \wedge r1 = 1 \wedge r2 = 0$?	

Figure 15: **lwsync**

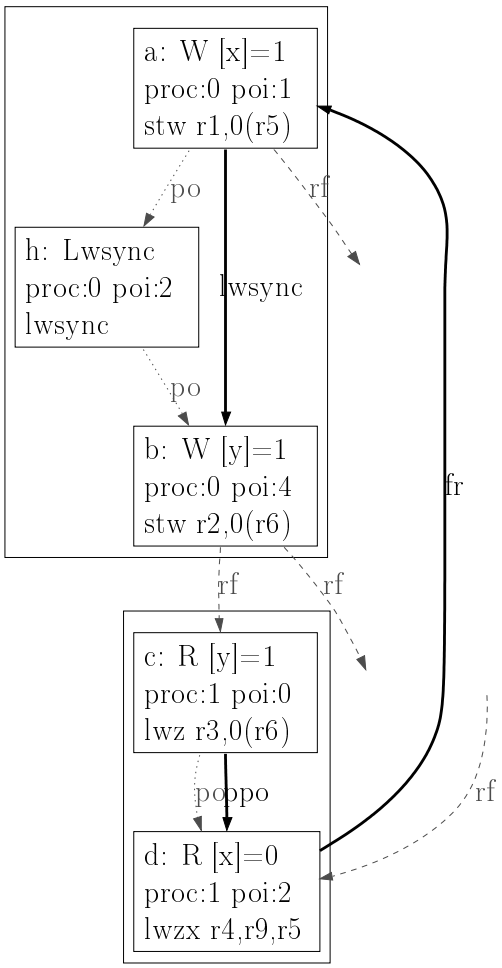


Figure 16: Not observed execution of **lwsync**

7 Conclusion

7.1 Contribution

We presented a generic framework for weak memory models at section 2, which includes formal definitions of *architectures* and *weak memory models*. We highlighted what we think to be the main concepts that should be precisely defined so as to provide formal weak memory models, namely globality of rfmaps and preserved program order.

In this framework, written in the Coq proof assistant [12], we have implemented classical formal models such as Sc and Tso which we proved equivalent to the native ones, namely SC [17] and Sparc TSO [1]. We provided a formal description of barriers power at section 3.1, together with a result on their placement in the code, which allowed us to retrieve the incremental description of Sparc TSO from two weaker models, which implementations we have sketched. Thus, we illustrated the power of barriers as restoring a memory model from a *weaker* one, as defined at section 2.

As described at section 2.4 we provide an executable version of our framework written in OCaml, memevents, which exhaustively outputs all *valid* – in a sense we defined formally at section 2 – executions of a test run on a particular architecture, together with a testing tool *litmus*, which allows us to run the same tests on particular hardware, so as to compare our model and a given machine.

These framework and tools, together we the testing methodology we provide at section 2.4 allowed us to provide a model for the Power architecture, which has been observed to be *valid* and *accurate* – in a sense that we defined at section 2.4 – with respect to the hardware.

Thus, we have provided a formal theory – together with precise vocabulary as required by recent work on memory models [14] – which we believe to have proved useful on current architectures, via our simulation and testing tools.

7.2 Status of writes

We consider, in our framework, the writes to be *non-atomic* as opposed to previous generic studies [11]. To do so, we do not impose that communication through memory – which are depicted by our $\overset{\text{rf}}{\rightarrow}$ relations – are visible to all processors.

Moreover, we rely on a particular notion of a write being *performed*, which is inspired by the *globally performed* notion from [15]. This is not the most fine grain notion of performed, as one may distinguish between *performed in storage* and *performed in memory* [19], or between *performed with respect to a processor* and *performed with respect to all processors* [5]. However, this has proved to be enough to highlight precise and fundamental notions to reason about several modern memory models, and to provide an accurate yet simple attempt at defining a model for the Power architecture including barriers semantics.

As underlined in the section 6.1, we believe we need to extend our framework so as to provide a model that is not only valid but also easy to program above, as well as to handle an accurate description of locks implementation as proposed in the Power documentation [5].

Acknowledgments

We thank Assia Mahboubi and Vincent Siles for help and advices on the Coq development. We thank Damien Doligez and Xavier Leroy for valuable discussions.

This work made use of the facilities of HPCx, the UK's national high-performance computing service, which is provided by EPCC at the University of Edinburgh and by STFC Daresbury Laboratory, and funded by the Department for Innovation, Universities and Skills through EPSRC's High End Computing Programme.

References

[1] The Sparc Architecture Manual Version 8, 1992. `http://www.sparc.org/standards/V8.pdf`.

[2] The Sparc Architecture Manual Version 9, 1994. `http://www.sparc.org/standards/V9.pdf`.

[3] *Power ISA Version 2.05*. October 2007. `http://www.power.org/resources/reading/PowerISA_V2.05.pdf`.

[4] *Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A*. Intel Corporation, March 2009. rev. 30.

[5] *Power ISA Version 2.06*. January 2009. `http://www.power.org/resources/reading/PowerISA_V2.06_PUBLIC.pdf`.

[6] Intel 64 Architecture Memory Ordering White Paper. August 2007.

[7] A. Adir, H. Attiya, and G. Shurek. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. *IEEE Transactions on Parallel and Distributed Systems*, May 2003.

[8] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. 1996.

[9] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January 2009.

[10] Alpha Architecture Reference Manual, Fourth Edition, 2002. `download.majix.org/dec/alpha_arch_ref.pdf`.

[11] Arvind and J.-W. Maessen. Memory model = instruction reordering + store atomicity. In *Proc. ISCA 2006*, June 2006.

[12] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, EATCS Texts in Theoretical Computer Science.

[13] H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.

[14] S. Burckhardt and M. Musuvathi. Memory model safety of programs. In *ECA-2*, 2008.

[15] M. Dubois and C. Scheurich. Memory access dependencies in share-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6), June 1990.

[16] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. *WRL Research Report*, 95(9), 1995.

[17] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.

[18] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL 2009*, January 2009.

[19] J. M. Stone and R. P. Fitzgerald. Storage in the PowerPC. *IEEE Micro*, 1995.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399